AD-A050 287 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/4
SEQUENTIAL AND PARALLEL PICTURE ACCEPTORS.(U)
DEC 77 A ROSENFELD AFOSR-77-3271
UNCLASSIFIED TR-613 AFOSR-TR-78-0119 NL

1 OF 2
AD A050287

2

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

2

TR-613
AFOSR-77-3271

December 1977

SEQUENTIAL AND PARALLEL PICTURE ACCEPTORS

Azriel Rosenfeld
Computer Science Center
University of Maryland
College Park, MD 20742

ABSTRACT

This report consists of drafts of Chapters 3-5 of a forthcoming book on "Picture Languages". [A draft of Chapter 2, on digital topology, was issued earlier this year as TR-542*.] The present chapters deal with sequential and parallel (= cellular) string and array acceptors. Future chapters will cover other acceptor models, as well as generator models (grammars). Comments on the choice and treatment of the material are invited.

The support of the Directorate of Mathematical and Information Sciences, U. S. Air Force Office of Scientific Research, under Grant AFOSR-77-3271, is gratefully acknowledged, as is the help of Mrs. Shelly Rowe in preparing this paper. The author also wishes to thank Chuck Dyer and David Milgram for many helpful discussions.

*References to "Section 2.x" can be found in Section "x" of that report.

19 **REPORT DOCUMENTATION PAGE**	READ INSTRUCTIONS BEFORE COMPLETING FORM
18 1. REPORT NUMBER AFOSR TR-78-0119 ✓ · 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 SEQUENTIAL AND PARALLEL PICTURE ACCEPTORS.	5. TYPE OF REPORT & PERIOD COVERED Interim rept.,
	6. PERFORMING ORG. REPORT NUMBER 14 TR-613 ✓
7. AUTHOR(s) 10 Azriel/Rosenfeld	8. CONTRACT OR GRANT NUMBER(s) 15 ✓AFOSR 77-3271✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Maryland ✓ Computer Science Center College Park, MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 16 17 61102F 2304 A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM ✓ Bolling AFB, DC 20332	12. REPORT DATE 11 Dec[redacted] [redacted]77
	13. NUMBER OF PAGES 174 12 166p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automata
Formal languages
Cellular automata
Picture languages

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report consists of drafts of Chapters 3-5 of a forthcoming book on 'Picture Languages'. The present chapters deal with sequential and parallel (= cellular) string and array acceptors. Future chapters will cover other acceptor models, as well as generator models (grammars). Comments on the choice and treatment of the material are invited.

403 018

Table of Contents

CHAPTER 3

STRING ACCEPTORS

3.1 Introduction

The purpose of this and the following chapters is to define various types of formal machines -- automata -- that can recognize given classes of pictures. Before defining these machines for two-dimensional arrays (i.e., digital pictures), we devote this chapter to reviewing the analogous concepts in one dimension, where the machines recognize given classes of strings or "tapes".

In order to make the analogy between the one- and two-dimensional cases more explicit, we have somewhat modified the standard definitions of the one-dimensional machines; but our definitions are readily equivalent to the standard ones. We also present a collection of propositions and theorems about one-dimensional machines; many of these are standard, while others are given as preparation for presenting their two-dimensional analogs. We give informal proofs of many of these results, but occasionally we state them without proof and give references to standard texts where proofs can be found.

The computational models described in this chapter are not all of equal utility; some of them are potentially of practical interest (e.g., bounded cellular automata), while others are much less so. However, even the impractical models are pedagogically useful, since they provide a standard framework for breaking computations down into small steps.

3.2 Automata and acceptors

3.2.1 Transition functions and configurations

Informally, a (one-dimensional) automaton can be thought of as a "bug" that, at any given time, is in one of a set of states, and is located at a given position on a tape that contains a sequence of symbols. The bug operates in a sequence of discrete time steps, at each of which it

a) Reads the symbol in its current position, erases that symbol, and replaces it by a (possibly) new one

b) Changes to a (possibly) new state

c) Moves to a neighboring position, or possibly does not move.

In general, the new state that the bug goes into, the new symbol that it writes, and the direction in which it moves all depend on its current state, on the current symbol, and on the direction in which the bug last moved. (The reason for making them dependent on the direction of last movement will be explained later.) Thus the bug's behavior is described by a transition function that maps (state, symbol, direction) triples into (state, symbol, direction) triples. We call the bug deterministic if each triple maps into a unique triple; otherwise, we call it nondeterministic, and we regard it as mapping triples into sets of triples.

On a more formal level, an automaton M is defined by specifying a triple (Q,V,δ), where

Q is the set of <u>states</u>

V is the set of <u>symbols</u> ("vocabulary")

δ: $Q \times V \times \Delta \rightarrow 2^{Q \times V \times \Delta}$ (or $\rightarrow Q \times V \times \Delta$, in the deterministic case) is the <u>transition function</u>

$\Delta \equiv \{L,R,N\}$ is the set of <u>move directions</u> ("left", "right", and "no move")

A <u>tape</u> is a mapping $\tau: I \rightarrow V$ from the integers into the vocabulary. A <u>configuration</u> of the automaton (and tape) is a quadruple (q,d,i,τ), where q is M's current state, d is the direction in which M has just moved, i is an integer denoting M's position, and τ is the current tape. The transition function δ defines a mapping μ between configurations in the following way: Let $\tau_i \epsilon V$ be the symbol in the ith position on τ. Let (q',v',d') be any triple in the image of (q,τ_i,d) under δ, where d and d' are directions in δ; in other words, suppose that when M is in state q, reads symbol τ_i, and has just moved in direction d, one of the possibilities for its behavior is that it rewrites τ_i as v', moves in direction d', and changes to state q'. Let τ' be the tape that is identical to τ except that τ_i has been changed to v' (i.e., $(\tau')_i = v'$). Then the configuration (q',d',i',τ') is a possible successor to (q,d,i,τ), where

$$i' = \begin{cases} i-1, & \text{if } d' = L \\ i, & \text{if } d' = N \\ i+1, & \text{if } d' = R \end{cases}$$

This formalism corresponds to the intuitive description of M given in the first paragraph.

3.2.2 Finiteness conditions

Up to now we have not imposed any restrictions on the sizes of the state set Q, the vocabulary V, or the tape τ. From now on we shall require that

> Q be a finite, nonempty set containing a special "initial state" q_0
>
> V be a finite, nonempty set containing a special "blank symbol" #
>
> τ be of the form $\#^{\infty}\sigma\#^{\infty}$, where σ is a finite, non-null string of non-#s

We shall also require that the initial position of the automaton M be on a non-# symbol.

An M that satisfies these restrictions will be called a Turing machine (TM). We shall also consider three special types of TM's:

a) #-preserving: M neither creates nor destroys #s -- in other words, for all $(q',v',d')\in\delta(q,v,d)$ we have v = # iff. v' = #.

b) Tape-bounded: M "bounces off" #s, so that it is essentially confined to the non-# portion of its input tape. Specifically, we require that if M has just moved in direction d and reads the symbol #, it must not rewrite the # as anything else, and must move in direction $d' = d^{-1}$, where $L^{-1} = R$ and $R^{-1} = L$. (We cannot have had d=N, since initially M is on a non-#, and it is easily shown by induction that whenever the move just made is N, the current symbol is not #.) Clearly tape-bounded implies #-preserving. Note that if M is tape-bounded, we can require τ to be of the form $\#\sigma\#$, since M never visits any other part of τ.

c) Finite-state*: M never rewrites any symbols -- i.e., $(q',v',d') \in \delta(q,v,d)$ implies $v'=v$. (In *particular, such an M is #-preserving.*)

The reason we required a transition of M to depend on the direction in which M has just moved was to allow us to define tape-boundedness as done here. If M's current move did not depend on the previous move's direction, there would be no way to guarantee that M bounces off #s; M would not know which way to move when it reads a #.

An alternative possibility for defining tape-boundedness would be to introduce two special "#" symbols, ℓ and r, and have ℓ's to the left of the non-# segment of the tape and r's to its right. We could then stipulate that whenever M reads an ℓ it moves right, and whenever it reads an r it moves left. This too would insure that M bounces off the #s. To generalize it to rectangular arrays of non-#s in two dimensions (Section 4.3) we could introduce two further special "#" symbols, t and b, at the top and bottom

*A better form would be "non-rewriting", but we use "finite-state" for historical reasons.

of the non-#s, and require that whenever M reads a t or b it moves down or up, respectively. However, this approach does not generalize to arbitrary (non-rectangular) two-dimensional arrays of non-#s (Section 4.4).

3.2.3 Acceptors and their languages

We now introduce the notion of an automaton's "accepting" its input tape. Formally, an acceptor A is a triple (M,q_0,Q_A), where $M = (Q,V,\delta)$ is an automaton, $q_0 \in Q$ is M's initial state, and $Q_A \subseteq A$ is a set of special "accepting states". We say that A accepts the tape τ_0 from position i_0 if repeated application of M's transition function, starting from the configuration (q_0,N,i_0,τ_0), leads to a state set containing some $q \in Q_A$. If M is a Turing machine (and is #-preserving, tape-bounded, or finite-state), we call A a Turing acceptor (TA), #-preserving acceptor (#PA), tape-bounded acceptor (TBA), or finite-state acceptor (FSA).

As defined so far, acceptance depends on the initial position i_0 of A. In the following propositions, we show that this position can be standardized without changing the set of tapes that are accepted. From now on, A is a TA with initial tape $\#^\infty\sigma\#^\infty$, and we shall speak of A as accepting the string σ if it accepts the tape $\#^\infty\sigma\#^\infty$.

Let $L_L(A)$ be the set of strings σ that A accepts when its initial position is at the left end of σ. Let $L_\cup(A)$ be the set of σ's that A accepts from some initial position; in other words, for all $\sigma \in L_\cup(A)$, there exists an initial position (on a symbol in σ) from which A accepts σ. Let $L_\cap(A)$ be the set of σ's that A accepts from any initial position (on any symbol in σ).

Proposition 3.2.1. For any A there exists an A' such that

$L_{\cap}(A') = L_{\cup}(A') = L_{L}(A)$. Moreover, if A is #P, TB, or FS, so is A', and if A is deterministic, so is A'.

Proof: Given A, we define A' as follows: From any initial position on σ, A' moves leftward, not rewriting any symbols, until it bounces off a #. As soon as this happens, A' enters the initial state of A, and thereafter its behavior is identical to that of A. Since σ is not changed by the initial movements of A', it is clear that A' accepts, from every initial position, exactly the same σ's that A accepts from their left ends. Moreover, the initial movements of A' do not violate #-preservation, tape-boundedness, or finite-stateness; hence if A has any of these properties, so has A'. Note also that if A is deterministic, so is A'.//

Proposition 3.2.2. For any A there exists an A" such that $L_{L}(A") = L_{\cup}(A)$. Moreover, if A is a #PA, TBA, or FSA, so is A".

Proof: Given A, we define A" as follows: From its initial position at the left end of σ, A" moves rightward, without rewriting any symbols. When it reads any non-# symbol, A" can nondeterministically enter the initial state of A, and thereafter its behavior is identical to that of A. Since its initial behavior has no effect on σ, it is clear that A" accepts σ iff. there exists an initial position from which A accepts σ. Clearly if A is a #PA, TPA, or FSA, so is A", but note that A" is nondeterministic even if A is deterministic.//

Proposition 3.2.3. For any A there exists an A* such that $L_L(A^*) = L_\cap(A)$. Moreover, if A is #P or TS, or is deterministic, so is A*.

Proof: A* rewrites each (non-#) symbol x of σ as a pair (x,x). Starting at the left end of σ, it then simulates A on the first terms of these pairs, leaving the second terms intact. If A rewrites a #, say as y, the simulation creates a pair of the form (y,#). If the simulation accepts, A* scans the non-#s and restores the pairs to their initial condition, i.e., it turns pairs of the form (y,#) into #s, and pairs of the form (y,x) into (x,x). A* then marks the left end of σ, and moves to the next symbol. From this starting point, it again simulates A on the first terms of the pairs; if the simulation accepts, it restores the pairs, finds the first unmarked symbol, marks it, and moves to the next symbol. The process described in the preceding sentence is repeated until every symbol is marked; if this happens, A* accepts. Clearly this will happen iff. σ is in $L_\cap(A)$. Evidently A* is #P, TB, or deterministic if A is.//

Propositions 3.2.1-3 imply that in most cases it does not matter whether we define acceptance from a standard position (e.g., the left end), from some position, or from all positions. [Instead of the left end, we can use any specific position, provided A can locate that position; e.g., we would not want to use the midpoint of σ as a starting point if A is FS, since an FSA cannot locate midpoints (see, e.g., Section 3.3.2.).] It will be convenient to use the left-end definition from now on. The set $L_L(A)$ will be called the <u>language</u> of A, and will be denoted by $L(A)$.

3.3 The language hierarchy

A language accepted by a #PA, TBA, or FSA will be called a #-preserving, tape-bounded, or finite-state language (#PL, TBL, or FSL). The classes of all such languages will be denoted by $L_{\#P}$, L_{TB}, and L_{FS}, respectively, and the class of all languages accepted by TAs (i.e., the class of TLs) will be denoted by L_T. We will use the prefix D if the acceptors are required to be deterministic -- e.g., L_{DTB} is the class of languages accepted by deterministic TBAs.

Determinism is not a restriction in the case of arbitrary TAs; it can be shown that $L_{DT} = L_T$. For the proof see [1, pp. 95-6]; given an arbitrary TA, A, we can define a DTA, A', that systematically simulates all possible sequences of transitions of A, so that A' accepts iff. A accepts.

It can also be shown that $L_{DFS} = L_{FS}$; the proof depends on the fact that any FSA can be simulated by a "one-way" FSA (see Section 3.3.3) that moves only from left to right. Given any one-way FSA, A, we can define a one-way DFSA, A', whose states are sets of the states of A, and that keeps track of all the states that A could possibly be in at a given step (see [1, pp. 31-2] for the details). We define the accepting states of A' as those sets that contain accepting states of A ; thus A' accepts iff. A does.

It is an open question whether $L_{DTB} = L_{TB}$ (and similarly for #P).

In the remainder of this section we establish some further inclusion relations among the classes of languages accepted by the various types of TAs.

3.3.1 $L_{\#P} = L_{TB}$

In this section we show that TBA's are as strong as #PA's -- i.e., they accept the same class of languages. We also show that tape-bounded FSA's are as strong as non-tape-bounded FSA's. The basic idea of the proofs is that when a #PA moves onto #s, its returning state(s), if any, can be predicted by an acceptor that never leaves the non-#s.

Let A be a #PA having $|Q|$ states, and suppose that A enters the #s by moving off the right-hand end of σ, while in some state $p \in Q$. If A is deterministic, then either it never returns, or it returns in some specific state $q \in Q$; if A is non-deterministic, it either never returns, or can return in any of a set of states $Q_R \subseteq Q$. We shall show that if A returns at all, it does so without ever getting farther away than $|Q|^2$ from σ; and in fact, if A can return in state q, it can return in that state without getting farther than $|Q|^2$ away from σ.

Let ρ be a legal path (i.e., a sequence of possible moves and state changes of A) on the #s that causes A to return to σ in state q. Suppose that the farthest ρ gets away from σ is $n > |Q|^2+1$, where n is as small as possible for any such ρ, and where the number of times that ρ gets n away from σ is also as small as possible. Thus ρ passes through the n #s to the right of σ, which we shall call $\#_1, \ldots, \#_n$. Suppose that ρ requires N time steps, which we shall denote by $t_1, \ldots, t_N$.

It is easily seen that there must exist two time steps $t_{i_1} < t'_{i_1}$ at each of which A is on $\#_1$, and between which it is to the right of $\#_1$ and visits $\#_n$ at least once. Similarly, during the time interval (t_{i_1}, t'_{i_1}), there must exist two time steps $t_{i_2} < t'_{i_2}$ at which A is on $\#_2$, and between which it is to the right of $\#_2$ and visits $\#_n$ at least once. Continuing this argument, we obtain a nested set of time step pairs

$$t_{i_1} < t_{i_2} < \cdots < t_{i_{n-2}} < t_{i_{n-1}} < t'_{i_{n-1}} < t'_{i_{n-2}} < \cdots < t'_{i_2} < t'_{i_1}$$

such that, for all $1 \le j \le n-1$, A is on $\#_j$ at t_{ij} and t'_{i_j}, is to the right of $\#_j$ between these times, and visits $\#_n$ at least once during that period.

Let q_j, q'_j be the states of A at time steps t_{i_j}, t'_{i_j} of ρ, $1 \le j < n-1$. Since $n-1 > |Q|^2$, two of the state pairs (q_j, q'_j) must be the same, say $(q_j, q'_j) = (q_k, q'_k)$, where $j<k$. For any time steps t and t', let $\rho(<t)$ denote the part of ρ prior to t, let $\rho(>t')$ denote the part subsequent to t', and let $\rho(t,t')$ denote the part between t and t' (inclusive). Now consider the path ρ' obtained by concatenating $\rho(<t_{i_j})$, $\rho(t_{i_k}, t'_{i_k})$, and $\rho(>t'_{i_j})$. Since $q_j = q'_j$, $q_k = q'_k$, and A stays to the right of its starting point between these two states, we see that ρ' is a legal path for A on the #'s, and it causes A to return to σ in state q after leaving it in state p. Moreover, during the part of ρ' corresponding to $\rho(t_{i_k}, t'_{i_k})$, A no longer reaches $\#_n$; in fact, it only gets $(n-k)$ to the right of $\#_j$, where $j+(n-k)<n$ since $j<k$. Thus ρ' is a legal path that either does not get as far as $\#_n$, or gets there fewer times than ρ did; and this contradicts our choice of ρ.

We have thus shown that if there exists any legal path for A that causes it to return to σ in state q after leaving it in state p, then there must exist such a path that never gets farther than $|Q|^2+1$ from σ. This observation allows us to prove

Theorem 3.3.1. $L_{\#P} = L_{TB}$.

Proof: Clearly $L_{TB} \subseteq L_{\#P}$, since TBAs are #PAs. Hence we need only show that for any #PA, A, there exists a TBA, A', that accepts exactly the same strings as A. Specifically, we define A' to simulate A as long as A remains on σ. If A leaves σ, say in state p, then by the above discussion, A' can determine whether A ever returns to σ in any given state q by internally simulating the behavior of A on the $|Q|^2+1$#s adjacent to σ. [Paths which take A farther than $|Q|^2+1$ away from σ can be ignored, since all return possibilities for A occur on paths that stay within $|Q|^2+1$ of σ, and if there are no such paths, A never returns. If A accepts without returning to σ, we can replace it by an A* that simulates it until it accepts, then moves back to σ and accepts; thus A* accepts exactly the same strings as A, but never accepts while on #s, and so can be simulated by A'.] Since the #PA (A or A*) need only be simulated on a string of #s of bounded length, its simulation by A' requires only a bounded amount of internal memory, and can be carried out every time A' bounces off a #. Thus A' can go into a non-accepting, absorbing state if A can never return; and into a returning state of A, otherwise, after which A' continues the simulation of A (unless acceptance has occurred). Clearly A' accepts iff. A accepts.//

In the proof of Theorem 3.3.1, if A is finite-state, so is A'; we thus have

<u>Theorem 3.3.2.</u> $L_{TBFS} = L_{FS}$.//

In the deterministic case, we can give a somewhat different proof of Theorems 3.3.1-2, based on the fact that when a deterministic #-preserving automaton moves onto #s, it suffices to simulate its behavior for $O(|Q|^2)$ <u>time steps</u> (rather than out to <u>distance</u> $|Q|^2$). As we shall see in Chapter 4, this deterministic proof generalizes, in part, to two dimensions; we therefore present it here.

<u>Proposition 3.3.3.</u> Let A be a deterministic FSA that has $|Q|$ states, and let the input tape of A be constant, say z^∞ for some $z \in V^*$. Then after less than $|Q|$ steps, the sequence of states of A becomes periodic with period $p \leq |Q|$.

<u>Proof</u>: No matter how A moves, it always reads the same symbol z; hence its next state depends only on its current state. Since A has only $|Q|$ states, some state must occur twice during the first $|Q|$ steps of A's operation. Suppose that the

*In this proposition we temporarily relax the requirement that the input tape must be of the form $\#^\infty \sigma \#^\infty$.

states at steps i and i+p are the same, where $0 \le i < i+p \le |Q|$. Then the sequence of states at steps $i, i+1, \ldots, i+p-1$ must be repeated at steps $i+p, i+p+1, \ldots, i+2p-1$, and the same sequence must then occur at steps $i+kp, i+kp+1, \ldots, i+(k+1)p-1$, for any $k \ge 1$, so that after time $i-1 < |Q|$, A's behavior is periodic with period (at most) $p \le |Q|$.//

Corollary 3.3.4. Let A be a determinsitic #PA that has $|Q|$ states. Then the behavior of A while it is on the # part of its input tape becomes periodic, with period $\le |Q|$, after $< |Q|$ steps.

Proof: Since A cannot rewrite #s, we can regard it as an FSA as long as it stays on #s.//

Proposition 3.3.5. Let A be as in Corollary 3.3 4, and suppose that A moves onto the # part of its tape. Then either A returns to the non-#s within $|Q|^2 + |Q|$ steps, or it never returns.

Proof: Suppose A leaves the non-# string σ at its right end. If it returns during the non-periodic part of its behavior (see Corollary 3.3.4), it has returned within $|Q|$ steps, and we are done. Otherwise, let its position at the beginning of its periodic behavior be r symbols to the right of σ; clearly $0 < r \le |Q|$. During a single period of its periodic behavior, let A's maximum leftward excursion from its starting position (at the beginning of the period) be s symbols, and let its net leftward excursion at the end of the period be t symbols, where evidently s and t each $\le |Q|$. If $s \ge r$, then A returns to the non-#s during its first period, i.e., within time $\leq 2|Q|$. If not, and if $t \leq 0$, A never returns to the non-#s. But if $t>0$, A is closer to the non-#'s at the end of each period than it was at the beginning, so that eventually it must return. In fact, it returns within at most k periods, where $r-kt \leq s$; this implies that k is at most $\frac{r-s}{t} \leq r \leq |Q|$. Thus the total time to return is at most $|Q|$ (the nonperiodic behavior) + $|Q|^2$ (at most $|Q|$ periods, each of length $\leq |Q|$).//

Corollary 3.3.4 and Proposition 3.3.5 provide us with a proof that $L_{D\#P} = L_{DTB}$ and that $L_{DFS} = L_{DTBFS}$. Indeed, for any D#PA,A, we can define a DTBA,A', that simulates A on σ. When A leaves σ, A' can immediately determine, by simulating a bounded number of transitions of A, whether or not A ever returns, and if so in what state, or whether A accepts without returning; A' can then continue the simulation of A, or can accept. Clearly A' accepts iff. A does, and if A is finite-state, so is A'.

3.3.2 $L_T \underset{\neq}{\supset} L_{TB} \underset{\neq}{\supset} L_{FS}$

It is well known that $L_T \underset{\neq}{\supset} L_{TB}$; in fact, one can prove a much more general proper inclusion result about a hierarchy of tape bounds (see [1, p. 149]). Thus we see that inability to create or destroy #s reduces the power of a TA. On the other hand, if we only prohibit creating #s, a TA's power is not reduced. In fact, given any TA, A, we can define an A' that simulates A, except that whenever A creates a #, A' creates a special symbol ♮, not in the vocabulary of A. For every transition of A when it reads a #, A' has both that transition and another one with the # replaced by ♮. Evidently (by induction on the number of elapsed steps) the configurations of A and A' are the same except that for A' some #s have become ♮s. Thus the sequences of states (or state sets) of A and A' are the same, so that A' accepts iff. A accepts.

We can also show that $L_{DTB} \underset{\neq}{\supset} L_{FS}$. For example, consider the set of strings σ of the form $x^n y^n$. A DTBA can accept this set by moving back and forth, alternately erasing x's and y's (changing them to z's, say) at the ends of the string, and verifying that the last y is erased just after the last x is erased. On the other hand, suppose this set were accepted by an FSA, call it A. We can assume without loss of generality that A starts at the left end of σ and accepts at the right end (given any A, simulate it by an A' that, as soon as A accepts, moves to the right end and then accepts). Thus there is a last step at which A crosses the boundary between the x's and the y's. At this step, A is in one of Q possible states, or $2^{|Q|}$ possible sets of states. But since there are infinitely many possible n's, the state (set) of A must be the same for two of them, say n_1 and n_2. Since A never returns to the x's, its further states do not depend on the numbers of x's; hence if A accepts $x^{n_1}y^{n_1}$, it must also accept $x^{n_2}y^{n_1}$, contradiction.

3.3.3 One-way acceptors

In this section we consider TAs that are allowed to move only in one direction, say to the right. Since we have assumed that a TA always starts out at the left end of its input string σ, such a "one-way" TA (OWTA) sees all of σ. Moreover, an OWTA may as well accept as soon as it sees a #, since from then on it will see nothing but #s, and can gain no further information about σ. (More precisely: let A be any OWTA, and let A' simulate A unitl it sees a #. At that point, A' can decide whether A, starting in its current state and reading only #s, can ever enter an accepting state ; if so, A' accepts, otherwise not.) Note that our OWTA thus always accepts, if at all, after $|\sigma|$ steps.

For OWTAs, there is no restriction in being deterministic, tape-bounded, or even finite-state. In fact, given any OWTA, A, we can define a deterministic OWTA, A', whose states are the sets of states of A, and show that A' accepts iff. A does (see the beginning of Section 3.3). The fact that a tape-bounded OWTA can simulate an arbitrary OWTA was pointed out in the previous paragraph. Finally, to see that a finite-state OWTA (A') can simulate an arbitrary one (A), note that the transitions of A cannot depend on the symbols that it writes, since it can never go back to read these symbols. Thus if we define A' to be the same as A except that it never rewrites any symbols, we see that the sequences of states of A and A' as they scan σ are the same, so that A' accepts iff. A does.

It is well known that $L_{OWFS} = L_{FS}$; for the proof see [1, pp. 41-44]. Thus the class of languages accepted by one-way acceptors of any type is just the class of finite-state languages.

3.4 Cellular acceptors

Up to now we have considered automata that operate sequentially by moving around on an input tape and reading one symbol at a time. In this section we introduce a class of parallel automata that operate on the entire tape simultaneously. We shall show that these automata, when regarded as acceptors, are exactly as strong as TBA's. On the other hand, we shall show that they are generally much faster than TBA's -- i.e., they can often accept in a much shorter time.

3.4.1 Cellular automata and acceptors

Informally, a cellular automaton is a string of "bugs" or "cells" each of which, at any given time, is in some state. The cells operate in a sequence of discrete time steps, at each of which every cell reads the states of its left and right neighbors, and changes to a (possibly) new state. Thus, formally, a cellular automaton K is defined by specifying a pair (Q,δ), where Q is the set of states, and δ is the transition function that maps triples of states into sets of states (or into single states, if K is deterministic), i.e., $\delta: Q^3 \rightarrow 2^Q$ (or $\rightarrow Q$). Here, for each cell $c \in K$, δ maps the triple (state of left neighbor, state of c, state of right neighbor) into the set of possible new states of c; in the deterministic case, this set always consists of a single element. A configuration of K is simply a mapping from the integers into Q which specifies the state of each $c \in K$.

We shall assume from now on that Q is finite, and that there is a special state $\# \in Q$ such that the initial configuration of K is of the form $\#^\infty \sigma \#^\infty$, where σ is a finite, non-null string of non-#s. If, in addition, δ is #-preserving (i.e., $\# \in \delta(q,r,s)$ implies $r = \#$, and $\delta(q,\#,s) = \{\#\}$ for all q,s in Q), we call K a bounded cellular automaton*. Note that in the bounded case we may as well assume that the string

*As in Section 3.3.2, inability to create #s is not a restriction on K. We shall assume hereafter that no K ever creates #s.

of cells is finite, and has the form $\#\sigma\#$, since the #s will always remain #s.

We now introduce the notion of acceptance of an input string (of states) by K. Formally, a <u>cellular acceptor</u> (CA), C, is a triple (K, Q_I, Q_A), where K is a cellular automaton, $Q_I \subseteq Q$ is a set of <u>initial states</u>, and $Q_A \subseteq Q$ is a set of <u>accepting states</u>, with $\# \in Q_I$. If K is bounded, we call C a <u>bounded cellular acceptor</u> (BCA). An input string is a configuration whose image is in Q_I, i.e., a string of the form $\#^\infty \sigma \#^\infty$ where $\sigma \in Q_I^+$. We say that C <u>accepts</u> this string (or, for brevity: that C accepts σ) if repeated application of K's transition function, starting from this configuration, can lead to an "accepting configuration". Such a configuration can be defined in a number of ways (compare Section 3.2.3):

a) Every $c \in C$ has a state in Q_A.

b) Some $c \in C$ has a state in Q_A.

c) A particular $c_0 \in C$ -- e.g., the one at the left end of σ -- has a state in Q_A.

Let $L_\cap(C)$, $L_\cup(C)$, and $L_L(C)$ be the sets of input strings that C accepts according to these three definitions. We will assume here that accepting states cannot be destroyed, i.e., for all $q,s \in Q$ and all $r \in Q_A$ we have $\delta(q,r,s) = \{r\}$. We also assume that accepting states cannot cause #s to be rewritten, i.e., that if q or s is in Q_A we have $\delta(q,\#,s) = \{\#\}$.

<u>Proposition 3.4.1</u>. For any C there exists a C' such that $L_L(C') = L_\cap(C)$. Moreover, if C is bounded or deterministic, so is C'.

Proof: The non-initial states of the cells in C' are pairs of the form (ξ, Q), where Q is a state of C. At the first time step, c_0 goes into state (1,q), where q is its initial state, and all other (non-#) cells go into states (0,q'), where the q' are their initial states. (This can be done since c_0 is the only cell whose left neighbor is #.) C' then simulates C by applying the transition function of C to the second terms of its states. If a cell c that has a # as a neighbor enters a state whose second term is in Q_A, c sends out a signal (using the first terms of the states) that spreads from cell to cell, provided that the cells involved have states whose second terms are in Q_A. If c_0 receives this signal from both sides, it accepts. Clearly this can only happen if the initial string was in $L_\cap(C)$. Note that this proof depends on our requirements that accepting states are never rewritten and cannot cause #s to be rewritten. If not for the first requirement, c_0 could never discover that, at a given time step, every non-# cell happens to be in a Q_A state; and if not for the second, the signals from c_0 might never catch up with the (receding) #s. We have also made use of our assumption that C never creates #s; otherwise the signals could never know that they have seen all of the non-# cells.//

Proposition 3.4.2. For every C there exists a C" such that $L_L(C") = L_\cup(C)$. Moreover, if C is bounded or deterministic, so is C".

Proof: We use pairs for the states of C", as in the proof of Proposition 3.4.1, and c_0 marks itself uniquely at the first time step. C" then simulates C using the second terms. If any cell's second term is in Q_A, that cell sends out signals (using the first terms) to the left and right; the propagation of these signals does not depend on the second terms. If a signal reaches c_0, it accepts. Clearly this can only happen if the initial string was in $L_{\cup}(C)$.//

Proposition 3.4.3. For any C" there exists a C such that $L_{\cup}(C)=L_L(C")$. Moreover, if C" is bounded or deterministic, so is C.

Proof: We again use pairs for the states of C, and c_0 marks itself uniquely (with first term 1) at the first step. Moreover, we define the accepting states of C as the pairs of the form (1,q), where $q \in Q_A$; pairs whose first terms are 0 are never accepting states. C then simulates C" on the second terms of its state pairs. The only cell of C that can ever enter an accepting state is c_0; thus $L_{\cup}(C)$ is trivially just $L_L(C")$.//

Proposition 3.4.4. For any C' there exists a C such that $L_{\cap}(C) = L_L(C')$, and if C' is bounded or deterministic, so is C.

Proof: We first define an automaton $\overline{C}'$ that simulates C' at odd-numbered time steps and whose states do not change at even-numbered steps. This is done using pairs of the form (ε,q)

propagation reaches that cell. Since the simulation of $\overline{C}'$ is quiescent at even-numbered time steps, the propagation must eventually catch up with it (even if $\overline{C}'$ is growing). Thus it is clear that if the simulation of $\overline{C}'$ accepts at c_0, then every cell of C accepts, and conversely, so that $L_{\forall}(C) = L_L(\overline{C}') = L_L(C')$. Note that if C' is bounded or deterministic, so is C.//

Prepositions 3.4.1-4 show that, in all cases, the classes of languages accepted by every cell, by some cell, and by a specific cell are the same. [Instead of the cell at the left end, we could use any cell that can uniquely identify and mark itself at the start of the processing.] We shall use the left-end definition of acceptance from now on, and we shall always assume that c_0 is uniquely marked.

The set $L_L(C)$ will be called the <u>language</u> of C, and will be denoted from now on by $L(C)$. Note that $L(C)$ can contain strings of any length, depending on how many cells of C were initially in non-# states. [C is still considered to be the same CA, no matter what the length of σ may be, as long as C has the same transition function.] The class of all languages accepted by (D)(B)CA's will be denoted by $L_{(D)(B)C}$.

for the states of $\overline{C}'$, where $\varepsilon = 0$ or 1 and q is a state of C'. At time step 0, each initial state q of a cell of C' is replaced by the state (1,q). Subsequently, we have $(0,q') \in \delta((1,q_1),(1,q),(1,q_2))$ iff. $q' \in \delta(q_1,q,q_2)$, and $\delta((0,q_1),(0,q),(0,q_2)) = \{(1,q)\}$ for all states q_1,q,q_2 of C'. Clearly the sequence of configurationsof $\overline{C}'$ at odd-numbered steps is the same as that of C' (at all steps) with (1,q)'s replacing q's. Thus if we define the accepting states of $\overline{C}'$ as the states (1,q) for which $q \in Q_A'$ is an accepting state of C', it is evident that the language of $\overline{C}'$, by any definition of acceptance, is the same as that of C'. Clearly if C' is bounded or deterministic, so is $\overline{C}'$.

We now define C to have as non-initial states pairs of the form $(\xi,\overline{q})$, where $\overline{q}$ is a state of $\overline{C}'$, and where c_0 is uniquely marked, as in the previous proposition (ξ=0 except for c_0, which has ξ=1). C simulates C' on the second terms of the pairs. We define the accepting states of C as all states of the form (2,q), for any accepting state q of $\overline{C}'$. If c_0 enters a state of the form (1,q) with q an accepting state of $\overline{C}'$, it immediately changes to state (2,q), irrespective of its neighbors' states. The states whose first terms are 2 then propagate, e.g., if any neighbor of cell c has state (2,q), then the next state of c is (2,q), irrespective of its own current state or its other neighbor's state. States whose first terms are 2 cannot enter into any other transitions, so that the simulation of $\overline{C}'$ stops at a given cell as soon as the

Appendix to Section 3.4.1

The assumptions that accepting states cannot be destroyed, and cannot cause #s to be rewritten, used in the proof of Proposition 3.4.1, can be eliminated, but at the cost of making the simulation very slow. The basic idea is as follows:

a) The simulation of C by C' proceeds one step at a time.

b) After each step, the simulation stops, and a signal is initiated from the ends of C that spreads through accepting states. If the signal reaches the distinguished cell c_0 from both sides, c_0 accepts; this can only happen if, at that step of the simulation, all non-# cells of C are in accepting states.

c) If a nonaccepting cell is found, the signal is modified to indicate this. When the modified signal reaches c_0, it initiates a synchronization process that causes every non-# cell of C' to change state in a special way simultaneously. When this happens, another step of the simulation of C is performed.

The synchronization is necessary to insure that every non-# cell of C' does its step of the C simulation at the same time.

In order to simulate C one step at a time, we use pairs of the form (ε,q) as states, where q is a state of C. For each transition $q' \in \delta(q_1,q,q_2)$ of C, we have a corresponding transition $(0,q') \in \delta(q_1,q,q_2)$ of the simulation. Thus initially, the first step of C is simulated, but the simulation stops at that point, since δ does not apply to pairs whose first terms are 0. We then carry out the signal propagation and synchronization steps, as described immediately below, using the first terms (ε) of the states. At the end of this process, the cells simultaneously change into the states which are the second terms of their pairs (i.e., $(\varepsilon,q) \to q$ for all ε), so that the next step of the C simulation can take place simultaneously at every cell. This step turns the cells' states back into pairs $(0,q')$, so that the simulation of C stops again until another signal propagation and synchronization has been performed. This process is repeated until the signal informs c_0 that all non-# cells of the simulation are in accepting states (second terms in Q_A); c_0 can then accept.

The signal propagation process is straightforward: If a cell c has a # neighbor and is in a state of the form $(0,q)$, where $q \in Q_A$, then c goes into state $(1,q)$; while if $q \notin Q_A$, c goes into state $(2,q)$. Subsequently, any cell c whose state is $(0,q_1)$ and which has a neighbor in state $(2,q_2)$ goes into state $(2,q_1)$; while if c has a neighbor in state $(1,q_2)$, then if $q_1 \in Q_A$, c goes into state $(1,q_1)$, while if $q_1 \notin Q_A$, c goes into state $(2,q_1)$. The distinguished cell c_0 is an exception: as long as either of its neighbors is in a state with first term 0, it does not change state. If both of its neighbors are in states with first term 1, and its own state is $(0,q)$ with $q \in Q_A$, it goes into state q (and thereby accepts). If one or both of its neighbors are in states with first term 2, or its own state is $(0,q)$ with $q \notin Q_A$, it goes into state $(3,q)$, which initiates the synchronization process to be described next.

Finally, the synchronization process operates as follows. Assume for simplicity that the distinguished cell c_0 is at the left end of the string σ of non-#s; if it is not, it can send a signal to the left end, which then acts as an initiator for the synchronization process. (See [2] for a synchronization construction that is somewhat faster and can proceed directly from an arbitrary initiation point.)

The left end sends two signals rightward, one of which travels three times as fast as the other; these signals are represented by special values of the first terms of the state pairs, say f and s. To transmit the fast signal, a cell whose left neighbor is # or f (we ignore the second terms from now on) becomes f, while an f becomes 0 at the next time step. To transmit the slow signals, a cell whose left neighbor is # or s becomes s"; an s" becomes an s', an s' becomes an s, and an s becomes 0 at the next time step. Thus the slow signal makes one move rightward at every third time step. The left end itself (3) change to 0 as soon as it has initiated the signals, so that the signals are only sent one.

When the f signal reaches the right end of σ, it is transformed into a left-moving fast signal f': A cell with f as its left neighbor and # as its right becomes f'; a cell whose right neighbor is f' becomes f', and f' becomes 0 at the next time step. The fast signal moves leftward until it meets the slow signal at the middle of σ. Specifically, if $|\sigma| = 2k+1$ is odd, then f' and s" would both reach the center cell (k+1) at time step 3k. Thus at step 3k-1, s is at cell k and f' at cell k+2, so that cell k+1 has s on its left and f' on its right. When this happens, we give cell k+1 the value x. Similarly, if $|\sigma| = 2k$ is even, f' reaches cell k+1 at step 3k-2, and s" reaches k at step 3k-3; thus at step 3k-2, s' and f' are adjacent at cells k and k+1. When this happens, we give both of these cells the value x. Thus when the signals meet we have 0's at all cells of σ except the midpoint(s), where we have x('s).

To illustrate the signal propagation process, we give two examples, for the cases $|\sigma| = 5$ and 6.

Step	$\lvert\sigma\rvert = 5$ (k=2)	$\lvert\sigma\rvert = 6$ (k=3)
0	f/s" - - - -	f/s"- - - - -
1	s' f - - -	s'f - - - -
2	s - f - -	s - f - - -
3	- s" - f -	- s"- f - -
4	- s' - - f'	- s'- - f -
5	- s - f'-	- s - - - f'
6	- - x - -	- - s"- f'-
7		- - s'f'- -
8		- - x x - -

We now carry out an analogous process in which each half of σ is bisected. An x sends fast and slow signals leftward and/or rightward through 0's; as soon as it has done so, it becomes a y, so that it does so once only. [If there are two x's, the left one sends signals leftward and the right one sends them rightward; if there is only one x, it sends them out on both sides.] The fast signals bounce off the ends of σ and meet the slow signals at the midpoints of their segments. Note that the two segments have the same length (k+1, if $|\sigma| = 2k+1$; k, if $\sigma = 2k$), so that this happens at the same time for both segments. When it happens, we again create x's; σ now has y('s) at its midpoint(s) and x's at its quarter and three-quarter points.

This process is repeated: the x's send out signals through 0's and become y's; the fast signals bounce off other y's or off the ends of σ, meet the slow signals, and create x's at the segment midpoints. Again, all these x's are created simultaneously. σ now has y's at its mid and quarter points, and x's at its 1/8, 3/8, 5/8, and 7/8 points, and we repeat the process again. The successive creations of x's are summarized below for the cases $|\sigma| = 11$ and 12.

Step	$\lvert\sigma\rvert = 11$	Step	$\lvert\sigma\rvert = 12$
15	- - - - - x - - - - -	17	- - - - - x x - - - - -
22	- - x - - y - - x - -	24	- - x - - y y - - x - -
24	x x y x x y x x y x x	26	x x y x x y y x x y x x

Eventually, the y's have only one or two 0's between them, so that when the signals meet, all the remaining 0's simultaneously turn into x's. Prior to this, no x or y ever had x's or y's on both sides of it; but now they all do. We can now use a transition that turns all cells whose first terms are x or y, and which have x's, y's, or #s on both sides, into their second terms, so that another step of C can be simulated; note that this transition applies simultaneously to every cell in σ.

The number of time steps required to create an x at the midpoint of a string of length 2k+1 is 3k+1, and the time to create x's at the midpoints of a string of length 2k is 3k. At each repetition of this bisection process, the length of the strings being bisected is less than half that at the previous repetition (2k+1 becomes k; 2k becomes k-1). Thus the total time required for the synchronization procedure is less than $\frac{3}{2}|\sigma|[1 + \frac{1}{2} + \cdots] = 3|\sigma|$. (This does not count the time needed for the chief cell to send a signal to the left end of σ to initiate the process, which is at most an additional $|\sigma|$ time steps.) As mentioned earlier, a more complicated synchronization procedure can be defined [2] which requires less than $2|\sigma|$ time steps; but in any case, the number of steps required is of order $|\sigma|$. Thus the simulation of a given C using this method requires $O(|\sigma|)$ time steps for each step of C, unlike the simulation in the proof of Proposition 3.4.1, which is nearly as fast as C itself, except for the propagation of the acceptance signal (which takes at most an additional $|\sigma|$ time steps).

3.4.2 Equivalence to sequential acceptors

Theorem 3.4.5. Turing acceptors can simulate cellular acceptors.

Proof: Given a cellular acceptor C with transition function δ, we define a Turing acceptor A that simulates C as follows: The vocabulary of A is the state set of C. To simulate one transition of C, A systematically scans σ, the non-# part of its tape, say from left to right. (Since #s are never created, A can know when this scan is finished.) At each step of this scan, A remembers (by storing them internally) the tape symbols at the two previous positions; in other words, if A is at position i, it remembers the symbols q_{i-2} and q_{i-1} at positions i-2 and i-1. A then reads the symbol q_i at position i, and rewrites it as $\delta(q_{i-2},q_{i-1},q_i)$ -- or, in the nondeterministic case, as some q in the set $\delta(q_{i-2},q_{i-1},q_i)$. A can then forget q_{i-2}, memorize q_i, and move on to position i+1. Thus when A has reached the right end of σ, say at position n, it has written in each position i a possible new state of the cell of C in position i-1. At this point, A remembers q_{n-1} and q_n and reads the # at the right of σ, so can compute a possible new state for the nth cell of C; it stores this state internally. [If C is not a BCA, A can write this state at position n+1, where there was formerly a #; and A can also write (in appropriate positions) the new states of the #s adjacent to σ, if C would have changed these to non-#s.] A now does a second scan, from right to left, and simulates the next transition of σ. This time, A can write the new states in their proper positions, rather than shifted one over to the right; e.g., in position n, A reads the current state of the cell in position n-1, and since A already knows the current states of the cells in positions n+1 (i.e., #) and n (stored internally), it can immediately write the new state for position n in that position. Thus in each scan of σ from left to right or from right to left, which takes $|\sigma|$ time steps, A can simulate a complete (parallel) transition of C. A accepts iff. its simulation of C accepts. Note that if C is deterministic, so is A, and if C is bounded, A is tape-bounded. Note also that this simulation is the fastest possible, since A cannot modify a tape symbol without visiting that position, and $|\sigma|$ time steps are needed to visit all of σ.//

Theorem 3.4.6. Cellular acceptors can simulate Turing acceptors.

Proof: Given a Turing acceptor A with transition function δ, we define a cellular acceptor C that simulates A as follows: The non-initial state set of C is $V \times (Q \cup \{0\}) \times D \times (D \cup \{0\})$, where V and Q are the vocabulary and state set of A, and $D = \{L,R,N\}$. At the first time step, each cell $c \in C$ except c_0 goes into a state of the form $(\beta,0,N,0)$, where $\beta \in V$ is c's initial state. Cell c_0 goes into the state (α,q_0,N,d), where $\alpha \in V$ is c_0's initial state, q_0 is A's initial state, and $d \in D$ is a direction in which A can move when it initially reads α (i.e., $\delta(\alpha,q_0,N)$ contains a triple whose third term is d).

At subsequent time steps, cells whose states and neighbors' states are of the form $(\beta,0,N,0)$ do not change state. Let c' be a cell whose state is of the form (α,q,d_1,d_2), and let (α',q',d_2) be a triple in $\delta(\alpha,q,d_1)$. Then

a) If $d_2 = N$, the new state of c' is of the form (α',q',N,d'), where $d' \in D$ is the third term of some triple in $\delta(\alpha',q',N)$. In this case the neighbors of c' do not change state.

b) If $d_2 = L$, the new state of c' is $(\alpha',0,N,0)$, and the new state of the left neighbor of c' is (β,q',L,d'), where $(\beta,0,N,0)$ was its old state, and where $d' \in D$ is the third term of some triple in $\delta(\beta,q',L)$. The right neighbor of c' does not change state.

c) If $d_2 = R$, the situation is analogous to that in (b), with R replacing L, and the roles of the left and right neighbors interchanged.

Evidently, at any time step there is exactly one cell c' whose state has the form (α,q,d_1,d_2), and that cell is in a position that A could have been in at the given time step.

[The term d_2 is needed in the state of c' so that its neighbors know which of them, if any, is to become the new c'. If c' did not pick its own (simulated) direction of motion, and if $\delta(\alpha,q,d_1)$ contained triples with more than one third term, then both neighbors of c' (or c' and one of its neighbors, or all three) might try to be come the new c'. This problem does not arise if A is deterministic, since in that case $\delta(\alpha,q,d_1)$ is always a singleton.]

This process is repeated until the cell c' enters a state (α,q,d_1,d_2) for which $q\in Q_A$, i.e., for which A would have accepted. At this point, c' sends an acceptance signal to c_0, and c_0 accepts as soon as it receives this signal. Thus C accepts iff. A does. Note that if A is deterministic, so is C, and if A is tape-bounded, we can regard C as bounded. [The latter requires special cases in the simulation to handle the steps in which A bounces off #s without C having to actually rewrite #s; the details will not be given here.] Note also that this simulation is the fastest possible, since C requires only one time step to simulate each time step of A.//

Theorems 3.4.5-6 immediately imply

<u>Theorem 3.4.7.</u> $L_{(D)C} = L_{(D)T}$; $L_{(D)BC} = L_{(D)TB}$.//

Thus we have not created any new classes of languages by introducing CA's.

3.4.3 Speed comparisons

As we have just seen, CAs can simulate TAs in real time; hence any language can be accepted by a (B)CA at least as fast as it can be accepted by a T(B)A. (This ignores the acceptance signal propagation process at the end of the simulation, which requires at most $|\sigma|$ time steps.) On the other hand, some languages can be accepted more quickly by (B)CAs than by T(B)As. In this section we discuss comparative acceptance speeds for sequential and parallel automata.

Let A (or C) be any acceptor, and suppose that there exists a function f, taking natural numbers into natural numbers, such that if A accepts the string σ at all, it accepts σ in at most $f(|\sigma|)$ time steps. We then say that A accepts <u>in time of order f</u>. For example, if f is linear, we say that A accepts in <u>linear time</u>; if f is the identity function, we say that A accepts in <u>real time</u>.

It is easily seen that, except in certain special cases, no A or C can accept in faster than real time. Indeed, suppose A accepts some string τ in m time steps, where $m < |\tau|$. Since A cannot make more than m rightward moves in m time steps, its state at time m cannot depend on the last $|\tau|-m$ symbols of τ. Thus A accepts <u>any</u> input string whose first m symbols are the same as those of τ. Similarly, suppose C accepts τ in $m < |\tau|$ steps. The state of c_0 at step m cannot depend on the initial states of cells farther than m away from c_0; hence C accepts <u>any</u> input string whose first m states are the same as those of τ. We may thus conclude that for "interesting" languages L, the time required to accept any $\sigma \in L$ must be at least $|\sigma|$.

There exist many simple languages which can, in fact, be accepted in real time. For example, let $L_{\bar{x}}$ be the set of strings in which a particular symbol x never appears. A deterministic FSA,A, that accepts $L_{\bar{x}}$ in real time can be defined as follows: A moves rightward; if it finds an x, it goes into a non-accepting state and stops moving; if it reaches a # without finding an x, it accepts. Similarly, a BCA,C, that accepts $L_{\bar{x}}$ in real time is defined as follows: An acceptance signal is propagated leftward from the right end of σ; if it meets an x, it stops; if it reaches c_0 without meeting an x, C accepts.

On the other hand, there are languages for which parallel acceptance is faster than sequential acceptance. An example is the set L_{sym} consisting of symmetrical strings of odd length made up of a's and b's, except for the center symbol, which is d. It can be shown [1, p. 145] that if A is any sequential acceptor whose language is L_{sym}, the time required for A to accept the string $\sigma \in L_{sym}$ of length 2n+1 is at least of order n^2. For example, we can define such an A informally as follows:

a) A memorizes the leftmost symbol of σ, erases it (i.e., rewrites it as some new symbol z), moves to the right end of σ and compares the rightmost symbol with the memorized symbol. If they are different, A stops moving and goes into a non-accepting state. If they are the same, A erases the rightmost symbol, memorizes the rightmost non-z symbol, moves to the left end of σ and compares the leftmost non-z symbol with the memorized symbol, and so on.

b) This process is repeated until a stage is reached where the symbol to be memorized is d rather than a or b. At this stage A checks that only z's remain on the other side of the d, and if so, it accepts.

Readily, acceptance by this A (which is a deterministic TBA) takes about $(2n+1) + (2n-1) + \cdots + 1 = (n+1)^2$ time steps.

A deterministic BCA,C, whose language is L_{sym} can be defined informally as follows:

a) An a or b that has the d as right neighbor changes to w; one that has the d as left neighbor changes to z. If the d has a as left neighbor and b as right neighbor, or vice versa, it changes to e; otherwise, it remains d.

b) a's and b's can shift rightward through w's and leftward through z's.

c) A z that has # as its right neighbor becomes an acceptance signal x that can propagate leftward through w's. If x reaches the d, it changes to a new signal y that can propagate leftward through z's.

Readily C accepts iff. its input string was in L_{sym} (otherwise the d would change to e at some point, or there would be excess a's or b's on one side of the d which did not change to w's or z's, and the acceptance signal could not reach c_0).

The time required for acceptance is about 3n, since it takes n steps for the first z to reach the right end, n steps for x to reach the d, and n steps for y to reach c_0.

We can consider upper, as well as lower bounds on acceptance time. A nondeterministic acceptor can take arbitrarily long to accept a given string, since it can make an unbounded number of nondeterministic transitions into non-accepting states before it finally makes one into an accepting state. Similarly, a non-(tape-)bounded acceptor can take arbitrarily long to accept, since it can perform an arbitrarily long computation before it accepts*. On the other hand, a TBA, if it accepts σ at all, must do so in less than $|\sigma+2||Q||V|^{|\sigma|}$ time steps. [Proof: there are only $|V|^{|\sigma|}$ possible tapes, on each of which the TBA can only be in $|\sigma+2|$ different positions (including its positions when it bounces off #s) and in $|Q-Q_A|$ nonaccepting states. If one of these configurations repeats, the configurations have become periodic and the TBA can never accept.] Similarly, if a BCA accepts σ at all, it must do so in less than $|Q|^{|\sigma|}$ time steps. (Using Proposition 3.3.5, we can similarly give an upper bound on the number of time steps within which a #PA must accept σ if it ever does so.)

---

*Indeed, given any time bound f, we can design our acceptor to first perform some operation that takes longer than $f(|\sigma|)$ time steps -- e.g., to compute $f(|\sigma|)^2$ -- before accepting.

In the remainder of this section we give several additional efficient algorithms for acceptance by CAs; all of these algorithms require time proportional to the string length $|\sigma|$.

a) Palindromes [3]: Let L be the set of symmetric strings $\sigma = \omega\omega^R$, where ω is a non-null string of a's and b's (say), and ω^R is the reveral of ω. To accept L, each cell of C sends a signal α leftward and rightward if its state is a, and a signal β leftward and rightward if its state is b. (These signals are erased when they reach the ends of the string.) If a pair of consecutive cells ever contain different signals (α and β or β and α), it cannot be the center pair of cells of a palindrome, and its cells mark themselves appropriately. Concurrently, the leftmost and rightmost cells send signals ℓ and r rightward and leftward. These signals meet at the center pair of cells. Evidently σ is a palindrome iff. this pair is unmarked when the ℓ and r signals meet there. (No other pair can remain unmarked, since for any other pair there is a # (whose occurrence is signaled by ℓ or r) located symmetrically to a non-#.) The ℓ signal is then erased, and the r signal becomes an acceptance or rejection signal (r' or r") which continues on to the left-hand cell; this cell then accepts or rejects, as appropriate.

b) Repetitions: Let L be the set of strings $\sigma = \omega\omega$, where ω is a non-null string of (say) a's and b's. To accept L, the rightmost cell of C sends out a signal α or β leftward, depending on whether its state is a or b. When this signal has passed it, the next-to-rightmost cell sends out an α or β signal leftward; when both these signals have passed it, the third-from-rightmost cell sends out an α or β; and so on. Thus the right end of σ generates a string of leftward-moving α's and β's representing the states of the cells in reverse order. Concurrently, the left end of σ sends out a signal ℓ rightward; this meets the string of α's and β's in the middle of σ. When they meet, the ℓ stops and becomes an ℓ'. As each α or β passes the ℓ', it becomes an α' or β'. When a cell in state a or b sees an α' or β', its state is erased, and the α' or β' becomes an x or y, depending on whether or not it matched (α' matches a, β' matches b). Evidently this process matches the signals from the cells in the right half of σ with the states of the cells in the left half of σ, rightmost first. If they all match, a string of x's is created; otherwise, some y's are also created. The cell c_0 at the left end receives this string; if it sees any y's, it rejects; if it sees no y's, and the x's stop coming, it accepts.

c) <u>Equal runs</u>: Let L be the set of strings σ of the form (say) $a^n b^n$, where $n \geq 1$. To accept L, C sends signals ℓ and r rightward and leftward from its ends. If ℓ sees anything but a's, it changes to ℓ'; if r sees anything but b's, it changes to r'. When ℓ and r meet, ℓ is erased, and r turns into r*, which continues leftward. If c_0 receives r*, it accepts; if it receives r', it rejects.

Acceptance of strings of the form $a^n b^n c^n$ is slightly more complicated [3]. Here signals are sent rightward and leftward from the ends, and they are also sent rightward from ab pairs and leftward from bc pairs. If the left end signal meets a bc signal at an ab boundary, and these signals have seen only a's and only b's, respectively, we know that σ begins with $a^h b^h$. Similarly, if the right end signal meets an ab signal at a bc boundary, and they have seen only c's and only b's, respectively, σ ends with $b^k c^k$. The left and right end signals can then continue until they meet; if they have seen only b's since meeting the bc and ab signals, we know that σ is of the form $a^h b^{h=k} c^k$. The left end signal can then be erased, and the right end signal can continue to c_0, which then accepts (or rejects, if one of the conditions failed to be met). We can similarly accept strings of the form $a^i b^j c^k$, where j=i <u>or</u> j=k, by verifying that at least one of the equality conditions is met.

d) <u>Parenthesis strings</u> [3]: Let L be the set of well-formed parenthesis strings σ (strings composed of several kinds of parentheses can be treated analogously). To accept L, each left parenthesis sends a signal α rightward, and each right parentheses sends a signal β leftward. The right end of σ also sends the signal r leftward. When an α meets a β, they cancel out. If a β reaches c_0, it rejects; this means that some initial segment of σ had an excess of right parentheses over left parentheses. If an α meets r, it changes to r', and if r' reaches c_0, it rejects; this means that some final segment of σ had an excess of left over right parentheses. If r reaches c_0, it accepts.

e) Counting [4]: Let L be the set of strings of (say) a's and b's in which the numbers of a's and b's are equal (or there are more a's than b's, etc.). C can accept a language of this type by counting the number of a's and b's and comparing the counts.

To count a's, we shift the a's leftward, and the cells of C simulate a binary counter with least significant bit at the left end. Specifically, the ith cell contains two bits (α_i, β_i), representing sum and carry bits, which are initially both 0. When an a reaches the left end, it is erased, and we add 1 to α_1 (i.e., if $\alpha_1=0$ we change it to 1; if $\alpha_1=1$ we change it to 0 and change β_1 to 1). The carry bits propagate rightward according to the following rules: whenever $\beta_i=1$, we change it to 0 at the next time step. At the same time, if $\alpha_{i+1} = 0$, we change it to 1; if $\alpha_{i+1} = 1$, we change it to 0 and change β_{i+1} to 1. It is easily seen that after at most $|\sigma|$ steps, all the a's have been shifted leftward and added into this counter; and after at most an additional $\log_2|\sigma|$ steps, the carry from the last addition has finished propagating.

To accept σ iff. the numbers of a's and b's are equal, or unequal in a specified order, C counts the a's and b's simultaneously (using two independent counters C_a and C_b). When a signal from the right end of C has reached the left end and returned to the right end, the counting must be finished. A second signal can then be sent out from the right end that compares the contents of the two counters, most significant bit first. If this signal finds that C_a and C_b match up to some point, and at that point C_a has 1 but C_b has 0, we know there are more a's than b's; while if C_a and C_b match all the way to the left end of σ, we know that the numbers of a's and b's are equal. This entire process takes linear time, namely, $3|\sigma|$ time steps.

3.5 Closure properties

In this section we show that the classes of languages defined in Section 3.3 are closed under various set-theoretic and geometric operations.

Proposition 3.5.1. A finite intersection of TLs, TBLs, or FSLs is a language of the same type.

Proof: Let $A_1,\ldots,A_n$ be acceptors for the languages $L_1,\ldots,L_n$ of the given type. Define A as follows: it first simulates A_1; if the simulation accepts, it simulates A_2; and so on, until it finally accepts if all the simulations accept. For TLs or TBLs, before carrying out the simulation we first replace each symbol by a pair of identical symbols, and then perform the simulation of A_1 on the second terms of the pairs, leaving the first terms intact; if A_1 accepts, we restore the original second terms (by setting them equal to the first terms) and then simulate A_2, and so on. Note that this construction preserves determinism.//

Proposition 3.5.2. A finite union of TLs, TBLs, or FSLs is a language of the same type.

Proof: Define A so it nondeterministically simulates one of the A_is; thus A accepts iff. one of the A_is would have accepted.//

For TLs and FSLs, the nondeterministic nature of this proof is unimportant, since determinism is no restriction (see the beginning of Section 3.3). For TBLs, to handle the deterministic case, we can use a different proof: Replace each symbol by an n-tuple of identical symbols, and then perform simulations of all the A_is simultaneously, using the ith term of each n-tuple to record the symbol that A_i would have read at that position, and also to record the location of A_i. At each step of the simulation, A scans the non-# part of the tape, finds each A_i and simulates its next move. A simulates the states of the A_is internally, using n-tuplesof states. If some simulation accepts, so does A. (Note that we can also use this proof for TLs, based on the fact that any TA can be simulated by a TA that never creates #s (Section 3.3.2), so that scanning the non-# part of the tape is still possible. We can also use n-tuples of states to simultaneously simulate all the A_is in the FSA case; here we can use the fact that it suffices to simulate one-way FSA's, so that we need not keep track of their locations, since they all move in step.) The same constructions can be used to prove Proposition 3.5.1, except that A accepts iff. all the simulations accept.

The complement of an FSL is also an FSL. This again follows from the fact that we need only consider one-way FSAs; given an OWFSA, A, that accepts L, we simulate it, and if it has not accepted at the end of its scan, we accept, so that evidently we accept exactly $\overline{L}$. The complement of a TL is not necessarily a TL; see [1, p. 123]. It is not known whether the complement of a TBL must be a TBL; see [1, p. 132].

<u>Proposition 3.5.3</u>. Any singleton $\{\sigma\}$ is an FSL.

<u>Proof</u>: Given $\sigma = s_1,...,s_n$, we can define a one-way deterministic FSA, A, that accepts σ. We do this as follows: If A

is in its starting state q_0 and reads s_1, it goes into state q_1;...,; if it is in state q_i and reads s_{i+1}, it goes into state q_{i+1};...; state q_n is an accepting state. If A is in any state $q_i (i\geq 0)$ and reads any symbol other than s_{i+1}, it goes into the non-accepting, absorbing state q_{n+1}. Clearly A accepts iff. its input is σ.//

Corollary 3.5.4. Any finite set of strings is an FSL.

Proof: Propositions 3.5.2-3.//

Let L_o be the set of strings that contain a given substring σ_o; then L_o is an FSL. Indeed, we can define a DFSA,A, that scans a given string from left to right, and whenever it sees the initial symbol x of σ_o, simulates an acceptor for $\{\sigma_o\}$. If this accepts, so does A; if not, A returns to that instance of x (which can be located because it is bounded distance away) and resumes its scan. [It follows by the preceding paragraph that the complement of L_o, i.e., the set of strings that fail to contain a given σ_o, is also an FSL. In particular, the set of strings that contain or fail to contain a given symbol is an FSL.]

A subset of an FSL is not necessarily an FSL. For example, the set of strings $\{x^i y^j | i,j\geq 1\}$ is evidently an FSL. (A one-way DFSA accepts it by staying in state q_0 as long as it reads x's; changing to state q_1 when it reads a y, and remaining in q_1 as long as it reads y's; and accepting if it reaches the right end of σ while in state q_1.) On the other hand, as we saw in Section 3.3.2, the subset $\{x^n y^n | n\geq 1\}$ of this language is not an FSL.

We now consider some geometric operations on languages. For any string σ, let σ^R denote the reversal of σ, i.e., if $\sigma=a_1 \ldots a_n$ then $\sigma^R=a_n \ldots a_1$. For any language L, let $L^R= \{\sigma^R|\sigma\in L\}$; we call L^R the reversal of L.

Proposition 3.5.5. The reversal of any TL,TBL, or FSL is a language of the same type.

Proof: Given an acceptor A for L, we can define an acceptor A' for L^R as follows: Starting at the end of an input string σ, A' moves to the right end, and then proceeds to simulate A with all move directions reversed, i.e., if A moves right, A' moves left, and vice versa. Clearly the successive configurations of A' are exactly the reversals of those of A, so that the simulation accepts σ^R iff. A accepts σ. Note that this construction preserves determinism, finite-stateness, and tape-boundedness.//

For any string $\sigma=a_1 \ldots a_n$, by a cyclic shift of σ is meant any string of the form $a_{k+1} \ldots a_n a_1 \ldots a_k$, for some $1\leq k\leq n$. (Note that for k=n this is σ itself.) For any language L, by the cyclic closure L of L is meant the set of all cyclic shifts of the strings in L.

Proposition 3.5.6. If L is a TL, TBL, or FSL, so is L.

Proof: We first consider the FSL case. Let A be a one-way DFSA that accepts L. Define an A' that starts at the left end of the given string σ in some state q of A, and moves rightward. At each move, A' computes two possible state changes for A, one corresponding to the symbol it actually sees, and the other corresponding to what would happen if A saw a #. If the latter possibility yields a non-accepting state of A, it is ignored. If it yields an accepting state, A' continues with two simulations, the original one and a new one in which A now enters its initial state q_0. The new simulation simply continues until the right end of σ is reached; but the original simulation continues to compute two alternatives at each step, and to create another new simulation whenever the # alternative yields an accepting state. This may happen many times, but the number of simulations that need be tracked is bounded, since A can only be in a bounded number of states. Thus at any given stage, A' is tracking the original simulation (which may keep on branching) together with a set of new simulations (which do not branch). When A' reaches the right end of σ, if any one of these new simulations is in the original state q, A' accepts; in fact, if this happens, σ is a cyclic shift of a string σ' in L, and the starting point of σ' is a point where A' began that particular new simulation. If none of the new simulations is in state q, A' can return to the left end of σ and begin the process again with a new starting state q'. If σ is in L, some starting state of A' at the left end of L must eventually lead to acceptance in this way. Note that this construction preserves determinism; alternatively, we could pick the starting state, and the point where A' switches to a new simulation, nondeterministically. Incidentally, we could have started A' at the left end simulating <u>every</u> state of A, and have A' keep track of a separate simulation for each starting state.

Next, let L be a TBL, and let A be a TBA for L. We define A' to move rightward until it marks some nondeterministically chosen point, and begins to simulate A, except that

a) Whenever the simulation tries to pass the marked point from the left, or reaches the marked point from the right, it behaves as though A had reached a #.

b) Whenever the simulation reaches an end of the string, and moves onto a #, it ignores the #, moves to the other end of the string, and behaves as if it had moved onto that string endpoint.

Evidently the successive configurations of this simulation (except while moving to another end of the string as in (b)) are essentially cyclic shifts of configurations of A on a string in L that ends at the marked point. Thus the simulation accepts iff. σ is a cyclic shift of a string in L.

If L is a TL, we must modify the construction in the preceding paragraph to allow simulation of rewriting #s. Specifically, we can modify (a) so that if the simulation rewrites a #, A' shifts (say) the initial segment of σ (up to the marked point) one space to the left, symbol by symbol, thus creating a space just after the marked point A' then writes the desired symbol in that space, moves the mark to that symbol, and resumes the simulation. [Alternatively, A' can simply create a new copy of σ in which the initial segment (up to the mark) now follows the final segment, and can then carry out the simulation on that copy.]//

For any language L, let L^+ denote the set of all concatenations of strings in L, i.e., $L^+ = \{\sigma_1 \ldots \sigma_n | n \geq 1;\ \sigma_i \in L,\ 1 \leq i \leq n\}$. We shall call L^+ the (concatenative) closure of L.

Proposition 3.5.7. If L is a TL, TBL, or FSL, so is L^+.

Proof: In the FSL case, we start at the left end of σ and simulate a one-way DFSA,A, for L, starting in its initial state q_0. At each point of σ we compute two possible state changes, as in the proof of Proposition 3.5.6. If the state change corresponding to a # does not yield an accepting state of A, we ignore it; but if it does, we continue with two simulations, the original one and a new one in which A reenters its initial state q_0. For both of these simulations, we compute two possible state changes at each point, and whenever the change corresponding to a # yields an accepting state of A, a new simulation branches off. This may happen repeatedly, but the total number of simulations that need to be tracked remains bounded by the state set size of A. When A' reaches the right end of σ and reads a #, if any of these simulations is in an accepting state, A' accepts. Evidently this happens iff. there exists a segmentation of σ into a concatenation of strings in L, where the breakpoints are just those points at which the simulation had switched to q_0.

Note that this is a deterministic construction; nondeterministically, we could simply pick an arbitrary set of points at which the simulation is in an accepting state, and switch back to q_0 at these points.

In the TBL case, we define A' to nondeterministically segment σ by placing a set of marks on it, and then to simulate an acceptor A for L successively on each of these segments σ_i, treating the marks as endpoints of σ_i (see (a) in the proof of Proposition 3.5.6). If each of these simulations accepts, A' accepts. Evidently this can happen iff. $\sigma \in L^+$. In the TL case the proof is similar, except that if the simulation needs to rewrite a #, A' shifts the initial or terminal part of σ in order to create a space at the appropriate segmentation point, as in the proof of Proposition 3.5.6.//

By the <u>concatenation</u> of the languages $L_1, \ldots, L_k$ we mean the set of strings $L_1 \cdots L_k = \{\sigma_1 \cdots \sigma_k | \sigma_i \in L_i, 1 \leq i \leq k\}$.

<u>Proposition 3.5.8</u>. Any concatenation of TL's, TBL's, or FSL's is a language of the same type.

<u>Proof</u>: The proof in the FSL case is analogous to that of Proposition 3.5.7, using one-way DFSA's $A_1, \ldots, A_k$ for the languages $L_1, \ldots, L_k$. At each branching we both continue the current simulation (of the A_i that accepted when it read a #) and begin a new simulation of A_{i+1}, starting in its initial state; thus we only allow sequences of branchings that are k long. At any stage, we may be simulating all k

A's, and for each of them, we may be in many different states, but the total number of cases to be tracked is still bounded (by the sum of the state set sizes of $A_1,\ldots,A_k$). When we reach the right end of σ, if a simulation of A_k is in an accepting state, A' accepts; evidently this happens iff. $\sigma \in L_1 \ldots L_k$. The proofs in the TBL and TL cases are just like those in Proposition 3.5.7, except that we segment σ into exactly k parts, and simulate the appropriate acceptor on each part.//

CHAPTER 4

SEQUENTIAL ARRAY ACCEPTORS

4.1 Introduction

Array acceptors are two-dimensional analogs of the automata studied in Chapter 3. Here the tape is an infinite two-dimensional array of symbols, and we shall usually require that all but finitely many of these symbols are #s. The non-#s may constitute a rectangular subarray, or we may only require them to be a connected subarray (see Section 2.2); both of these restrictions are generalizations of the requirement, in Section 3.2.2, that a one-dimensional tape must be of the form $\#^{\infty}\sigma\#^{\infty}$, so that the non-#s constitute a connected string. We will first study array acceptors in the case of a rectangular subarray (Section 4.3), and will then consider the generalization to an arbitrary connected subarray (Section 4.4). The rectangular case is more realistic, since digital pictures are rectangular arrays, but the connected case poses some interesting research problems. Sequential acceptors will be treated in this chapter; cellular acceptors will be studied in Chapter 5.

4.2 Array automata and acceptors

Informally, a two-dimensional automaton is a "bug" that, at any given time, is in one of a set of states, and is located at a given position on an array of symbols. Just as in the one-dimensional case, the bug operates in a series of discrete time steps, at each of which it

a) Reads the symbol in its current position, erases that symbol, and replaces it by a (possibly) new one

b) Changes to a (possibly) new state

c) Moves to a neighboring position, or possibly does not move.

Note that we could define "neighboring" here to mean one of the four horizontal or vertical *neighbors*, or we could allow it to include the four diagonal neighbors as well (see Section 2.2). For simplicity, we will always use the first of these definitions in this chapter. Note that a diagonal move can be accomplished by making one horizontal move followed by one vertical move (or vice versa); thus an automaton that is allowed to make diagonal moves can be simulated, at half the speed, by one that is allowed to make only horizontal or vertical moves.

In general, the state that the bug goes into, the symbol that it writes, and the direction in which it moves all depend on its current state, on the current symbol, and on the direction in which it last moved (see Section 3.2). Thus the bug's behavior is described, exactly as in the one-dimensional case, by a transition function that maps (state, symbol, direction) triples into (state, symbol, direction) triples; the only difference is that there are now five possible directions (left, right, up, down, and no move), rather than three (left, right, and no move). We call the bug deterministic if each triple maps into a unique triple; otherwise, we call it nondeterministic, and we regard it as mapping triples into sets of triples.

Formally, an array automaton M is a triple (Q,V,δ), where Q,V,δ are defined exactly as in the one-dimensional case (Section 3.2.1), except that $\Delta = \{L,R,U,D,N\}$ ("left," "right," "up," "down," and "no move"). A tape is a mapping $\tau: I\times I \to V$ from the pairs of integers (=coordinates of array points) into the vocabulary. A configuration is a quintriple (q,d,i,j,τ), where q,d,τ are as in Section 3.1.1, and (i,j) are the coordinates of M's position. The mapping μ between configurations induced by the transition function δ is defined as follows: Let $\tau_{ij} \in V$ be the symbol in the (i,j) position on τ, let $(q',v',d') \in \delta(q,\tau_{ij},d)$, and let $\tau'=\tau$ except that τ_{ij} has been replaced by v'; then $(q',d',i',j',\tau') \in \mu(q,d,i,j,\tau)$, where

$$(i',j') = \begin{cases} (i-1,j) & \text{if } d=L \\ (i+1,j) & \text{if } d=R \\ (i,j-1) & \text{if } d=D \\ (i,j+1) & \text{if } d=U \\ (i,j) & \text{if } d=N \end{cases}$$

As in Section 3.2.2, we shall require Q and V to be finite, nonempty sets, where Q contains a special "initial state" q_0, and V contains a special "blank symbol" #. We shall also assume that only a finite number of tape symbols are non-#s, where $\# \in V$. (Further restrictions on the set of non-#s will be considered in a moment.) If these restrictions hold, we call M a <u>Turing machine</u> (TM). If M cannot create or destroy #s, we call it <u>#-preserving</u> (#P); if M "bounces off" #s [i.e., $(q',v',d') \in \delta(q,\#,d)$ implies $v'=\#$ and $d'=d^{-1}$, where $L^{-1}=R, R^{-1}=L, U^{-1}=D, D^{-1}=U$], we call M <u>tape-bounded</u> (TB); and if M never rewrites any symbols, we call it <u>finite-state</u> (FS). There need be no confusion with the one-dimensional definitions, as long as we know which is meant.

As in Section 3.2.3, we define an <u>array acceptor</u> as a triple $A=(M,q_0,Q_A)$, where M is an array automaton, q_0 is M's initial state, and Q_A is a set of "accepting states." We say that A accepts the tape τ_0 from position (i_0,j_0) if repeated application of M's transition function, starting from the configuration (q_0,N,i_0,j_0,τ_0), leads to a state set containing some $q \in Q_A$. The special types of acceptors (Turing, #P, TB, FS) are also defined just as in Section 3.2.3. If Σ is the (finite) set of non-#s in τ_0, we may also speak of A as accepting Σ.

In Chapter 3 we required that the initial tape τ of a Turing machine be of the form $\#^{\infty}\sigma\#^{\infty}$, where σ is a finite, non-null string of non-#s. As pointed out in Section 4.1, there are two possible generalization of this requirement to two dimensions -- namely, we can require that the set Σ of non-#s be rectangular in shape, or merely that Σ be connected. In Section 4.3 we will assume that Σ is rectangular; we will consider the non-rectangular case in Section 4.4.

Before we begin our study of rectangular array acceptors, we show that an acceptor can verify that its non-# input array Σ (which we assume to be connected) is, in fact, rectangular. Indeed, we show that a deterministic, tape-bounded, finite-state array acceptor can do these things; thus all of the stronger types can certainly do so.

Proposition 4.2.1. There exists a DTBFSA, A, that accepts a connected array Σ of non-#s iff. Σ is rectangular.

Proof: A moves leftward until it bounces off a #, then upward until it again bounces off a #; if Σ is rectangular, this puts A in the upper left corner of Σ. A now operates as follows:

a) Move down. If that point is #, move back up and continue with step (b). If not, move left and check that a # is present. If it is not, reject; if it is, move right and up, and continue with step (b). [This verifies that the left border of Σ is locally straight at its upper end.]

b) Move up. If that point is not #, reject; if it is #, move back down, move right, and repeat the process. Continue until the rightward move hits a #. [This verifies that the top border of Σ is straight.] Move back and continue with step (c).

c) Move down. If that point is #, move back up and continue with step (d). If not, move right and check that a # is present. If if is not, reject; if it is, move left and up, and continue with step (e). [This verifies that the right border of Σ is locally straight at its upper end.]

d) Move left, move down, and check that a # is present. If it is not, reject; if it is, move back up and repeat the process. Continue until the leftward move hits a #. [This verifies that the bottom border of Σ is straight.] If this step is ever reached, accept.

e) Move left until a # is hit. Move down. If that point is #, move back up and continue with step (f). If not, move left and check that a # is present. If it is not, reject; if it is, move right and up, and continue with step (g). [This verifies local straightness of the left border.]

f) Move right, move down, and check that a # is present. If it is not, reject; if it is, move back up and repeat the process. Continue until the rightward move hits a #. [This verifies that the bottom border is straight.] If this stage is ever reached, accept.

g) Move right until a # is hit. Move down. If that point is #, move back up and continue with step (d). If not, move right and check that a # is present. If it is not, reject; if it is, move left and up, and continue with step (e). [This verifies local straightness of the right border.]

It is easily seen that A accepts Σ iff. Σ is rectangular. Note also that in the process of accepting a rectangular Σ, A does a raster scan of Σ, row by row (alternately left-to-right and right-to-left.//

4.3 Rectangular array acceptors

4.3.1. Rectangular array languages

If Σ is rectangular, it is easy to show, just as in Section 3.2.3, that in most cases the set of Σ's accepted by the acceptors of a given type is the same whether we require acceptance from all initial positions, from any initial position, or from some standardized initial position such as the upper left-hand corner of Σ. Specifically, given an acceptor A, let $L_\cap(A)$, $L_\cup(A)$, and $L_C(A)$ be the sets of Σ's accepted by A from all positions, from some position, and from the upper left corner of Σ, respectively. Then we can prove

Proposition 4.3.1. *For any A there exists an A' such that* $L_\cap(A')=L_\cup(A')=L_C(A)$; *if A is #P, TB, or FS, or is deterministic, so is A'.*

Proof: The proof is analogous to that of Proposition 3.2.1. From its initial position, A' moves leftward until it bounces off a #, and then upward until it bounces off a #, which puts it in the upper left corner of Σ; it then simulates A.//

Proposition 4.3.2. *For any A there exists an A" such that* $L_C(A")=L_\cup(A)$; *if A is #P, TB, or FS, so is A".*

Proof: The proof is analogous to that of Proposition 3.2.2. From its initial position in the upper left corner of Σ, A" moves nondeterministically rightward and/or downward, and at any step (provided it is reading a non-# symbol), it can begin to simulate A. [Alternatively, A" can do a systematic scan of Σ (see the proof of Proposition 4.2.1), and at any non-# step, can nondeterministically begin simulation of A.]//

Proposition 4.3.3. For any A there exists an A* such that $L_C(A^*) = L_\cap(A^*)$; if A is #P, TB, or deterministic, so is A*.

Proof: Here again the proof is analogous to that of Proposition 3.2.3: A* does a systematic scan of Σ; simulates A (on first terms of pairs) starting from each point of the scan; if the simulation accepts, A erases the traces of the simulation, marks the starting point, moves on to the next (unmarked) point of the scan, and repeats the process.//*

By these propositions, it makes no difference, in most cases, whether we define acceptance from a standard point such as the upper-left corner (or any point that A can locate), from all points, or from some point. We shall use the upper-left-corner definition from now on. The set $L_C(A)$ will be called the language of A, and will be denoted by $L(A)$. [Incidentally, as we shall see in Section 4.4, Propositions 4.3.2 and 4.3.3 hold for arbitrary connected Σs, but Proposition 4.3.1 must be modified for such Σs.]

*If A is not #P, erasing the traces of the simulation may not be trivial, since in the course of it, Σ may become nonrectangular. However, in Section 4.4.1 we shall show that a DTBA can in fact scan an arbitrary connected Σ.

4.3.2. The language hierarchy

Just as in the one-dimensional case, a rectangular array language accepted by a (deterministic) #PA, TBA, or FSA will be called a (D)#PL, (D)TBL, OR (D)FSL, respectively. The classes of all such languages will be denoted by $L_{(D)\#P}$, $L_{(D)TB}$, or $L_{(D)FS}$, and the class of all languages accepted by (deterministic) TA's will be denoted by $L_{(D)T}$. In this section we will establish some relationships among these classes of languages.

Determinism is not a restriction in the case of arbitrary TA's; in other words, $L_{DT}=L_T$. This can be shown in essentially the same way as in the one-dimensional case (see Section 3.3): Given an arbitrary TA, A, we can construct a DTA, A', that systematically simulates all possible sequences of transitions of A, so that A' accepts iff. A does. It is an open question whether $L_{DTB}= L_{TB}$ (and similarly for #P), just as it is in one dimension.

In one dimension, we had $L_{DFS}=L_{FS}$, but this is false in two dimensions. The proof of the following theorem is based on [5]:

Theorem 4.3.4. $L_{FS} \underset{\neq}{\supset} L_{DFS}$

Proof: Consider the set of square arrays Σ, having odd side length, composed of 0's and 1's. Note that a DFSA can recognize whether or not an array has these properties. [It can test for squareness by moving diagonally -- e.g., alternately down and right, beginning at the upper left corner, and verifying that it hits the bottom just at the lower right corner. It can test oddness of side length by moving along one side of the array and counting modulo 2; and it can verify that the array consists of nothing but 1's and 0's by doing a raster scan.] We want to accept an array of the above type iff. its center symbol is 1.

A nondeterministic FSA can do this as follows: Starting at the upper left corner, move diagonally downward and to the right. At some point, nondeterministically chosen, memorize the symbol just read on the diagonal, and begin moving diagonally downward and to the left. If this reaches the lower left corner (i.e., it hits the bottom just at the corner), then the memorized symbol was at the center point of the array; if it is 1, the FSA accepts. Evidently, acceptance occurs iff. the center point is 1.

To prove that a DFSA cannot do it, we first consider the behavior of such an FSA (call it A) relative to an m-by-m block (i.e., subarray) of 0's and 1's. A can enter the block at any one of 4m-4 positions, and can be in any one of $|Q|$ states when it enters. (It can also have just moved in either of two directions, if it enters at a corner; but this does not appreciably affect our argument.) Similarly, A can leave at 4m-4 places, in one of $|Q|$ states; or it can fail to leave, but we shall ignore this for the moment. The block thus defines a mapping from the $4(m-1)|Q|$ incoming (position, state pairs) into the $4(m-1)|Q|$ outgoing pairs. There are $[4(m-1)|Q|]^{4(m-1)|Q|}$ such functions. On the other hand, there are $2^{(m^2)}$ possible blocks, and for sufficiently large m we have $[4(m-1)|Q|]^{4(m-1)|Q|} = 2^{4(m-1)|Q|\log[4(m-1)|Q|]} < 2^{(m^2)}$. Thus there must be two different blocks B_1 and B_2 that give rise to exactly the same mapping.

To complete the proof, suppose that we had an A that could accept the arrays whose center points are 1's. We can assume that A does its accepting in a standard position, say at the lower right corner (given any A, we can simulate it until it accepts, then move to that corner and accept). Let (i,j) be a position in which $B_1=1$ and $B_2=0$ (or vice versa). Construct an array Σ that contains B_1 as a subarray, with its (i,j) point in the center of Σ_1; the rest of Σ_1 can be all 1's (say). When A accepts Σ_1, it starts and ends outside B_1. Hence A must also accept the array Σ_2 obtained from Σ_1 by replacing B_1 by B_2; but the center point of Σ_2 is 0, contradiction.//

In one dimension (Section 3.3.1), we saw that $L_{(D)\#P} = L_{(D)TB}$ and $L_{(D)TBFS} = L_{(D)FS}$. We can show this in two dimensions for deterministic acceptors; the nondeterministic case is still open (see [6]).

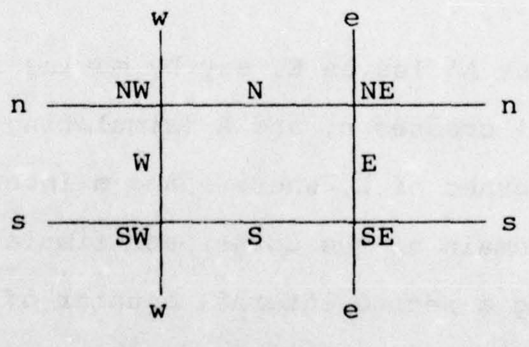
Note first that Proposition 3.3.3 and Corollary 3.3.4 hold in two dimensions as well as in one: The behavior of a DFSA on a constant input tape becomes periodic after at most $|Q|$ steps (where $|Q|$ is the number of states), with period at most $|Q|$, and the same is true for a D#PA while it is on the # part of its tape. We can also use these results to prove a two-dimensional analog of Proposition 3.3.5:

Lemma 4.3.5. Let A be a D#PA that has $|Q|$ states, and suppose that A moves onto the # part of its tape, say from the right edge of the non-# array Σ. Let c be the column (of IxI) that contains this edge of Σ. Then either A returns to c within $|Q|^2 + |Q|$ steps, or else it never returns.//

We can now prove

Theorem 4.3.6. $L_{D\#P} = L_{DTB}$; $L_{DFS} = L_{DTBFS}$.

Proof: Given any D#PA, A, we can construct a DTBA, A', that simulates A as long as it remains on Σ. When A leaves Σ, it must do so at one of the edges, and must enter one of the regions N, E, W, or S (see the diagram below), say region E.



Let n,s be the rows of IxI that contain the top and bottom of Σ, and let e,w be the columns of IxI that contain the right and left columns of Σ. Thus when A enters E, it does so from column e.

By Lemma 4.3.5, A either never returns to e, or it returns in at most $|Q|^2 + |Q|$ steps. In particular, if A ever gets farther than $\frac{1}{2}(|Q|^2+|Q|)$ to the right of e, it can never return to e (and hence can never return to Σ). Thus A' can remain on e, starting at the point where A left e, and keep track of the state changes of A using its own states, and of the left/right moves of A using an internal counter of capacity $\frac{1}{2}(|Q|^2+|Q|)$. If this counter overflows without A having returned to e, A' can stop the simulation, since A will never return. (Acceptance by A without returning will be discussed later.) At the same time, A' can simulate the up/down moves of A on the #s by moving up and down on e. As long as A remains in E, A' remains in Σ and no problems arise. If A returns to e (so that the counter becomes empty), A' is guaranteed to be in the proper state and at the point of e where A returns, and it can resume simulating A on Σ.

Suppose that A' leaves E, say by moving into NE. When this happens, A' crosses n, and A (simulating A' on e) reaches the northeast corner of Σ, where e and n intersect. At this point, A' can remain at the corner and simulate the up/down moves of A using a second internal counter of capacity $\frac{1}{2}(|Q|^2+|Q|)$. By the analog of Lemma 4.3.5, if this counter ever overflows without A having returned to n, A will never return, and the simulation can stop. Thus as long as A remains in NE and does not move too far away from Σ to return, A' can stay at the corner and simulate all the moves of A using its two counters. If A leave NE, it must enter either E or N, and A detects this by noting that one of the counters is empty. If A enters E, the up/down counter is empty, and A' resumes simulating the up/down moves of A by moving up and down on e, as before. Similarly, if A enters N, the left/right counter is empty, and A' now simulates the left/right moves of A by moving left and right on n, as long as A remains in N.

The procedure is analogous if A enters any of the other # regions. Thus wherever A moves onto the #s, as long as it remains close enough to Σ that return is still possible, A' can simulate A in such a way that if A does return, A' is in the proper position and in the proper state to resume simulating A on Σ.

If A accepts Σ without returning to it, this must happen within $|Q|$ steps, since the behavior of A on the #s is periodic with period at most $|Q|$. Thus this will always happen before the counters of A' have overflowed. If A' detects the fact that A has entered an accepting state, A' can accept immediately, without continuing the simulation. Thus in any case, A' accepts iff. A does, which proves the first part of the theorem.

Note finally that in our simulation of A by A', if A is FS, so is A', which proves the second part of the theorem.//

We can now also prove

<u>Theorem 4.3.7.</u> $L_{(D)T} \supsetneq L_{TB}$; $L_{DTB} \supsetneq L_{DFS}$

<u>Proof</u>: We know that these results are true in one dimension (see Section 3.3.2). Let S be a set of strings that is in L_T but not in L_{TB} (in one dimension). If we regard S as a set of n-by-1 arrays, then clearly no two-dimensional TBA can accept S, since this TBA remains on one row and can thus be simulated by a one-dimensional TBA. On the other hand, a two-dimensional (D)TA can accept S by simulating a one-dimensional TA that accepts S.

Similarly, let S be a set of strings that is in L_{DTB} but not in L_{FS} (in one dimension), and regard S as a set of n-by-1 arrays. If S were in L_{DFS} in two dimensions, it would be in L_{DTBFS} (Theorem 4.3.6), so that a two-dimensional DTBFSA would accept it; but this DTBFSA could be simulated by a one-dimensional DTBFSA, contradiction.//

4.3.3. Three-way acceptors

We can also define two-dimensional analogs of the one-way acceptors treated in Section 3.3.3. Specifically, let us require that A start at the upper left corner of its input array, and allow A to move right, left, and down, but not up. Such an A can still do a systematic scan of its array, but it can no longer verify that the array is rectangular. [In fact, A cannot even tell whether two consecutive rows have the same length; if it verifies that they line up at the left end, it can no longer do so at the right end.] We shall call such an A a three-way acceptor (TWA).

Evidently, on an n-by-1 array (confined to a single row), a TWA of a given type (Turing, #P, TB, FS) has the same power as a one-dimensional acceptor of the corresponding type, and can be simulated by such an acceptor. (If the TWA ever leaves that row, it can never return to it, so reads nothing but #s from then on, and can be simulated by a one-dimensional acceptor that remains on the row.) Thus any proper inclusion that holds for the languages of one-dimensional acceptors also holds for TWA languages.

On the other hand, on a 1-by-n array (confined to a single column), a TWA has the same power as a one-dimensional (D)FSA, by the same argument as in Section 3.3.3. It follows that a (four-way) two-dimensional acceptor of any type stronger than an FSA is strictly stronger than a three-way acceptor of the corresponding type. This is also true for FSA's, as we see from

Proposition 4.3.8. $L_{(D)FSA} \underset{\neq}{\supset} L_{(D)TWFSA}$

Proof: Consider the set of n-by-2 (i.e., two-row) arrays, say of 0's and 1's; we want to accept such an array iff. the two rows are identical. A DFSA can verify this by systematically checking that each symbol on the top row has an identical symbol below it. On the other hand, a TWFSA, A, that has $|Q|$ states must leave the top row in one of $|Q|$ states at one of n positions. For a given top row ρ, let S_ρ be the set of (state, position) pairs in which A leaves ρ. [If A is deterministic, S_ρ is a singleton.] If the bottom row is ρ, at least one such pair (q,i) must result in acceptance (after A has examined the bottom row). Thus (q,i) cannot be in $S_{\rho'}$ for any $\rho' \neq \rho$, so that each S_ρ contains a pair that is in no other S_ρ. But there are only $n|Q|$ pairs, and there are 2^n different rows, so this is impossible.//

We can also show, as in one dimension, that $L_{(D)TW\#P} = L_{(D)TWTB}$ and $L_{(D)TWFS} = L_{(D)TWTBFS}$. In fact, let A be a TW#PA, and let A' be a TWTBA that simulates A on each row. [On any given row, as long as A does not move down, A' can simulate it by the argument in Section 3.3.1.] If A moves down while on Σ, A' also moves down. Moreover, while simulating the moves of A off Σ, A' can determine in what states A can move down, and if the simulation enters such a state, A' can move down to the end of the row below and continue the simulation. As long as A does not move below the bottom row of Σ, A' can still determine in what states A can return to Σ, as in Section 3.3.1. If A does move below the bottom row, A' can stop the simulation and determine whether A can accept while on #s. In this simulation, if A is deterministic or finite-state, so is A'.

On the other hand, nondeterministic TWFSAs are strictly stronger than deterministic, and in fact, nondeterministic TWTBAs are strictly stronger than deterministic:

Proposition 4.3.9. $L_{TWFS} \supsetneq L_{DTWFS}$; $L_{TWTB} \supsetneq L_{DTWTB}$.

Proof: Consider the set of arrays of 0's and 1's that contain two vertically adjacent 1's. A nondeterministic TWFSA can accept this set by scanning its input array systematically, and when reading a 1, nondeterministically moving down and accepting if it finds another 1; evidently it accepts iff. there exists two vertically adjacent 1's. On the other hand, let A be a DTWTBA having $|Q|$ states. Then A must leave the top row of its input array at one of n positions and in one of $|Q|$ states. Since there are 2^n different top rows, there must exist two top rows R_1, R_2 that A leaves in the same state and at the same point. Let j be a position in which R_1 and R_2 differ; say R_1 has a 1 in the jth position, while R_2 has a 0. Consider the two-row array whose top row is R_1 or R_2 and whose bottom row is all 0's except for a 1 in the jth position. Then if A accepts the first of these two-row arrays, it also accepts the second.//

4.4 Connected array acceptors

4.4.1. Traversal theorems

If Σ is an arbitrary connected array, the dependence of acceptance on the initial position of the acceptor becomes a more complicated problem; the difficulty is that location of a standard initial position by the acceptor is no longer so easy. In fact, it is not even obvious that an acceptor from a given starting point can visit all of Σ, so that the acceptor may not even be able to reach the standard position. We therefore begin this section by proving some results about the ability of array automata of various types to traverse their input arrays.

Proposition 4.4.1. There exists a DTA that, from any starting point P on its input array Σ, can scan Σ completely and return to P with all marks made on Σ in the course of the scan erased.

Proof: Starting at P, the DTA, A, moves in a rectangular "spiral," e.g., in the sequence

16	15	14	13	12
17	4	3	2	11
18	5	P	1	10
19	6	7	8	9
20	21	...		

and marks the points of the spiral, whether they are non-# or #. Moving in a spiral is accomplished as follows: At any step, A is in a rightward, upward, leftward, or downward moving state. When it is in a rightward moving state, at every step it checks to see whether the point above it is marked, and if so, moves right. If the point above it is unmarked, it stays at that point and goes into an upward moving state. The behavior of A while in the other three types of states is analogous: when it is in an upward (leftward, downward) moving state, it checks the point on its left (below it, on its right) at each step; if this point is marked, it moves up (left, down), but if not, it stays at that point and goes into a leftward (downward, rightward) moving state.

Since Σ is finite, there will eventually be a complete turn of the spiral which finds only #s. A detects this event using an internal counter that is initially set at 0 and can count up to 5. Whenever A changes direction and is at a #, it adds 1 to the counter, but resets the counter to 0 whenever it reads a non-#. Thus the counter reaches 5 iff. A has changed direction four consecutive times without seeing a non-#. At this point, the marked points form a rectangle R that entirely contains Σ. (No point P' of Σ can be outside R, since there would have to be a path of non-#s between P' and P, and this path would have to cross the border of R.) Thus A has now seen all of Σ. If desired, A can now retrace the spiral in reverse and erase all the marks (or at least erase them from the non-#s, if A is not allowed to create #s).

When A finishes retracing the spiral, it has returned to P.//

Proposition 4.4.1 describes a relatively simple method of scanning Σ that can be implemented by a DTA because of its ability to move on #s and to mark them. A DTBA can also scan its input array Σ, but it must use a more complicated method, as we shall next see.

Theorem 4.4.2. There exists a DTBA that, from any starting point on its input array Σ, can scan Σ completely and return to P with all marks made in the course of the scan erased.

Proof: The DTBA, A, can use a graph traversal algorithm to scan Σ; see [7] for the graph version. A uses five marks, u, v, x, y, and z, which are mutually exclusive but which do not interfere with the original symbols of Σ. Initially, it marks P with u, and then proceeds as follows:

a) If the current point Q has an unmarked non-# neighbor, mark Q with x and move to such a neighbor, say Q'. If not, go to step (b).

a1) If Q' has a neighbor marked u, mark Q' with y, move back to Q (it is the unique neighbor of Q' marked x), mark Q with u, and return to step (a).

a2) If Q' has no neighbor marked u, mark Q' with z, move back to Q, erase all y marks from neighbors of Q, mark Q with u, move back to Q' (it is the unique neighbor of Q marked z), mark Q' with u, and return to step (a).

b) If Q has no unmarked non-# neighbor, erase all y marks from neighbors of Q and mark Q with v.

b1) If Q has a neighbor marked u, go to such a neighbor and return to step (a).

b2) If not, stop.

We observe first that step (b2) must be reached eventually. In fact, the algorithm cannot stay in step (a) indefinitely, since at each pass through (a) a previously unmarked point Q' of Σ is marked either y or u, and this can only happen finitely many times. Moreover, at each entry into step (b) some point Q is marked v, and this can only happen finitely many times.

Next we show that at each entry into step (a) the points marked u form a path $Q_1,\ldots,Q_k$ $(k \geq 1)$ with Q_1=P, Q_k=Q, and such that the path does not cross or touch itself, i.e., Q_j is a neighbor of Q_i iff. $j=i\pm1$. This is clearly true at the first entry into (a), since P was initially marked u and there are as yet no other marks. Furthermore, if those properties are true at a given entry into (a), they are still true at the next entry. Indeed,

a1) If the next entry is via (a1), the set of u's is unchanged (Q was initially marked u, (a) changed its mark to x, and (a1) changed it back to u).

a2) If the next entry is via (a2), a new point Q' marked u is added to the path; Q' has no neighbor marked u, other then Q after its x is changed back to u, so the path still does not touch or cross itself.

b1) If the next entry is via (b1), the mark on Q is changed to v, and the new Q is a neighbor of Q marked u (if any); by induction hypothesis, this Q ($=Q_{k-1}$) is unique, so the path has been shortened by one point, and still does not touch or cross itself.

Evidently (b2) can hold only if the path has length 1 ($Q=Q_k=Q_1=P$), since otherwise Q does have a neighbor marked u; thus when the algorithm stops, the current point is P, and there are no more points marked u.

Finally, we show that when the algorithm stops, every point of Σ has been marked v. Note first that when it stops, there can be no points marked x, y, z, or u, and there is at least one point (P) marked v. Suppose there were an unmarked point; let Q be such a point whose distance form P, as measured by the length of a shortest path in Σ from P to Q, is as small as possible. Thus Q has a neighbor Q' (the previous point on the path) that is closer to P than Q, so that Q' must be marked v. Let Q" be the neighbor of Q that last got marked v by the algorithm. By (b), just before Q" was marked v it had no unmarked neighbors. Since Q cannot have been marked u (u's must eventually be changed to v's), it must have been marked y at that time. But by (a1), for Q to have been marked y it must have had a neighbor Q* other than the current node (Q") that is marked u. Thus when Q" was marked v, Q still had a neighbor Q* marked u. This u eventually was changed to v by the algorithm, so that Q" was not the last neighbor of Q to be marked v, contradiction.

The foregoing discussion shows that the algorithm, starting at P on an unmarked array Σ, stops at P with every point of Σ marked with v. It follows that we can modify the algorithm, by interchanging the roles of points marked v and unmarked points, so that when it starts at P on an array having every point marked v, it stops at P with every point unmarked. Thus the algorithm together with this modification of it define the desired DTBA, and the proof is complete.//

Another DTBA traversal theorem will be given in Section 4.4.2, when we show that a DTBA can distinguish the outer border and the hole borders of its input array Σ, and can find (e.g.) the leftmost of the uppermost points of the outer border.

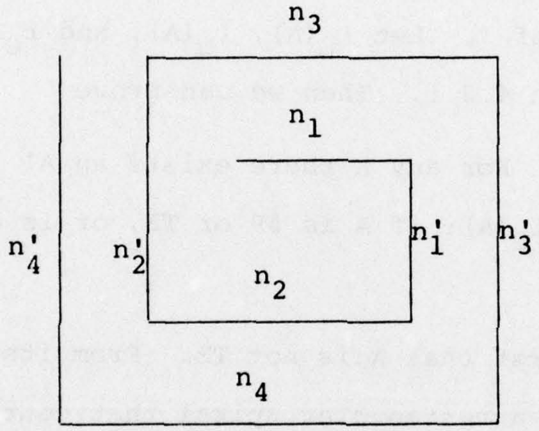
A nondeterministic TBFSA can traverse all of Σ by simply moving nondeterministically from neighbor to neighbor; but it can never know when it has finished, i.e., it can never correctly make a state change that depends on its having seen all of Σ, and it cannot recognize its starting point if it moves too far away from that point. The proof of the following result is based on [8].

Theorem 4.4.3. There exists no TBFSA that, from any starting point on any input array Σ, accepts when and only when it has visited all of Σ.

Proof: Suppose A were such a TBSFA, say having $|Q|$ states. Let us first consider the behavior of A on n-by-1 (i.e., one-row) arrays Σ_n that are surrounded by #s except possibly at their ends. A can enter such an array at either end in any of $|Q|$ states, and it can leave at either end in any of $2^{|Q|}$ sets of states. Thus the number of possible mappings from entering end and state to leaving end and state set is $(2\cdot 2^{|Q|})^{2|Q|}$. Since this number does not depend on n, there must be many n's that yield the same mapping, and in particular, there must be at least one infinite set of n's, say $N=\{n_1,n_2,n_3,\ldots\}$, with $n_1 < n_2 < n_3 \ldots$, that yield the same mapping. A similar argument, using 1-by-n (one-column) arrays Σ'_n shows that there exists an infinite set $N'=\{n'_1,n'_2,\ldots\}$ of lengths all of which yield the same input/output mapping for A.

Since for any Σ, A accepts iff. it has seen all of Σ, this is true in particular when Σ is a hollow rectangle one point thick. Let us choose such a rectangle R that has horizontal side length $n_i \in N$ and vertical side length $n'_j \in N'$. By the definitions of N and N', the behavior of A on R is the same no matter which $n_i \in N$ and which $n'_j \in N'$ we use to define R, in the sense that if A starts in state q_0 at some point of R, the (sets of) states in which A enters and leaves the various sides of R are the same for all such R's. Since A accepts such an R in a finite time, A can only make a finite number m_1 of clockwise circumnavigations of R, and a finite number of m_2 of counterclockwise circumnavigations, relative to its starting point, before accepting, and these bounds are valid for all such R's.

Let S be the rectangular spiral constructed from a succession of horizontal and vertical line segments of lengths $n_1, n_1', n_2, n_2', \ldots, n_{4m}, n_{4m}'$, where $m=\max(m_1,m_2)$; e.g., the first two turns of S might look like this:



(If two consecutive n_i's or n_j''s differ by 1, we can skip one of them to insure that the turns of S never touch one another.) Suppose that A starts out in state q_0 from a position on the mth turn of S, i.e., from one of the segments $n_{2m-1}, n_{2m}, n_{2m-1}'$, or n_{2m}'. By construction of S, the (sets of) states in which A enters or leaves the various sides of S are the same as those for the R's, as long as A never gets to an end of S. But in fact, A accepts the R's after having made at most $\max(m_1,m_2)$ turns around R, so that in fact A never reaches an end of S. Thus A accepts S without ever having seen its ends, contradiction.//

4.4.2. Connected array languages

We can apply the traversal results obtained in Section 4.4.1 to show that the sets of Σ's accepted by the arbitrary or the tape-bounded acceptors is usually the same whether we require acceptance from all initial positions, from any initial position, or from some standard initial position such as the "upper left corner" (=leftmost of the uppermost points) of Σ. Let $L_{\cap}(A)$, $L_{\cup}(A)$, and $L_{C}(A)$ be defined as in Section 4.3.1. Then we can prove

Proposition 4.4.4. For any A there exists an A' such that $L_{\cap}(A') = L_{\cup}(A') = L_{C}(A)$; if A is #P or TB, or is deterministic, so is A'.

Proof: Suppose first that A is not TB. From its initial position, A' marks a rectangular spiral that contains Σ, as in the proof of Proposition 4.4.1. By moving to the upper left corner of this spiral and then doing a row-by-row scan, A' can easily find the upper left corner of Σ. A' then simulates A, but ignoring the marks on the points of the spiral. Clearly A' accepts iff. A does, and if A is deterministic, so is A'.

To handle the case where A is #P or TB, we must show that there exists a DTBA that can find the upper left corner of its input array. This is shown in the proof of Theorem 4.4.5 immediately below.//

If A is FS, the proof of Proposition 4.4.4 breaks down; in fact, there does not exist a TBFSA that can find the upper left corner of its input array. Indeed, suppose A' were a TBFSA that accepted when it found this corner. Construct the hollow rectangles R and rectangular spiral S for A' as in the proof of Theorem 4.4.3, where the number of turns of S is chosen to exceed the number of turns that A' makes on R before accepting at R's upper left corner. Then if we start A' in the middle of S, it accepts at some north-west corner of S without ever having found the real upper left corner of S.

<ins>Theorem 4.4.5</ins>. There exists a DTBA that, from any starting point on its input array Σ, can find the upper left corner of Σ.

<ins>Proof</ins>: The proof makes extensive use of the topological properties of arrays studied in Chapter 2, and in particular, of the border-following algorithm BF defined in Section 2.6. We first show that our DTBA, A, can find the <ins>outer border</ins> of Σ; this is the border along which Σ is adjacent to the background component of $\overline{\Sigma}$, as distinguished from <ins>hole borders</ins> along which Σ is adjacent to other components of $\overline{\Sigma}$ (if any).

To find the outer border of Σ, A operates as follows:

a) A moves up until it bounces off a #.

b) When this happens, A has hit a border of Σ. A now marks its position, say with the mark α, and proceeds to follow the border, using essentially the border-following algorithm BF of Section 2.6. (A can evidently check the neighbors of the α by making a sequence of local moves, and determine which of these neighbors is the next border point specified by BF.) The point α may be on several borders of α, but the border that A follows is the one defined by the component of #s containing the # just above α.

c) During border following, A keeps track of its net up/down moves by moving another marker, β, along the border. Specifically, whenever A moves down, it marks its current position on the border (say with γ), follows the border back to β (or to α, if β has not yet been used), erases β, and rewrites it one step further along the border. A then returns to γ, erases it, and resumes following the border. [Note that β can never catch up with γ, since the number of downward moves made by A cannot exceed its total number of moves along the border.] Similarly, whenever A moves up, it marks its position with γ, goes back to β, erases it, rewrites it one step further back along the border, returns to γ, erases it, and resumes border following. The markers β and γ must contain information that specifies which # neighbor of the marked point is the current # neighbor being used in the BF algorithm; this is because the border may pass through some points twice, and β and γ must be able to distinguish between two positions along the border even if they are at the same point of Σ.

d) If β is in the same position as α when A comes to move it further back, or if A moves up before β is created, the current position of A on the border (as marked by γ) is higher up than the point at which A hit the border (as marked by α). If this happens, A erases α and β, returns to γ, erases γ, and goes back to step (a).

e) At each entry into step (a), A is higher up than it was at the previous entry; hence steps (a-d) cannot keep repeating indefinitely, since Σ is finite. Thus eventually, while executing steps (b-c), A will get all the way around the border and back to α without case (d) occurring. This means that α is at a highest point of the border. Now the highest points of a hole border are evidently the points of Σ just above the highest points of the hole, and cannot have points of the hole above them. Since α does have above it a # belonging to the component of #s whose border A has been following (see steps (a-b)), this component cannot be a hole. Thus when case (d) fails to occur, α is on the outer border of Σ, and indeed, on an uppermost point of this border.

We must now show that A can find the leftmost of the uppermost points on the outer border of Σ. To this end, A can mark its starting point with α and proceed to follow the outer border, keeping track of its net downward moves by moving a marker β along the border, as in (c) above, and marking its current position with γ. Whenever β gets back to α, the net downward displacement of A is zero, so that γ is at an uppermost point. A marks the position of γ at that time with δ, and resumes border following. When the border has been completely followed, and A gets back to α, the uppermost points of the border have all been marked with δ's (or α).

To find which of these points is the leftmost, A once again follows the outer border, this time using the mark β to keep track of its net leftward or rightward moves. (A uses an internal state to tell it whether β is counting leftward or rightward moves.) Whenever A comes to a δ, if β is counting rightward moves, this δ must be to the right of the α, so A just continues to follow the border; but if β is counting leftward moves, A changes the δ to an ε, and marks the current position of β with a ζ, before resuming border following. The next time A comes to an δ, if the β is counting rightward moves, or is counting leftward moves and is between the α and the ζ, A just continues border following, since this δ either is to the right of α or is not as far to the left of α as the ε is; but if the β is counting leftward moves and is farther from the α than the ζ is, A changes the ε back to δ, changes the current δ (where γ is located) to ε, erases the ζ, puts an ζ at the current position of β, and then resumes border following. When A has followed the border completely around and returned to α, the ε will be at the leftmost of the δ's, i.e., at the upper left corner of Σ. If desired, A can then go around the border again, erase all the other marks, and follow the border around to the ε.//

[A modification of the proof of Theorem 4.4.5 is used in [9] to construct a DTBA, T, that systematically scans its input array by finding the outer border, following it, and scanning each row of Σ, going around hole borders as it encounters them. While scanning a row ρ, say from left to right, when T hits a hole border, say at λ, it follows that border around, keeps count of its net up-down moves, and marks with μ's all the intersections of the border with ρ. One of these intersections is closest to λ on its right, and so is the next point (after λ) in which ρ meets Σ. T can identify this point by keeping count of net left/right moves, and can then resume scanning ρ starting from this point.]

Returning to the relationship between $L_\cap$, $L_\cup$, and L_C, we next prove

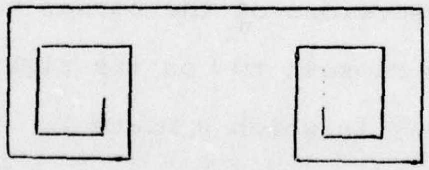
Proposition 4.4.6. For any A there exists an A" such that $L_C(A") = L_\cup(A)$, and if A is #P, TB, or FS, so is A".

Proof: Analogous to the first proof of Proposition 4.3.2.//

Proposition 4.4.7. For any A there exists an A* such that $L_C(A^*) = L_\cap(A)$, and if A is #P, TB, or deterministic, so is A*.

Proof: Analogous to that of Proposition 4.3.3, using Proposition 4.4.1 or Theorem 4.4.2.//

The analog of Proposition 4.4.7 for TBFSA's is false. Let L be the language consisting of all arrays Σ that fail to contain a particular symbol x. Evidently we have $L=L_\cap(A)$, where A is the TBFSA that accepts from state q_0 if it reads any symbol other than x; thus A accepts Σ from every starting position iff. Σ contains no x's. On the other hand, suppose that we had $L = L_C(A^*)$ for some TBFSA A*. In analogy with the proof of Theorem 4.4.3, we can construct a double rectangular spiral S of the form

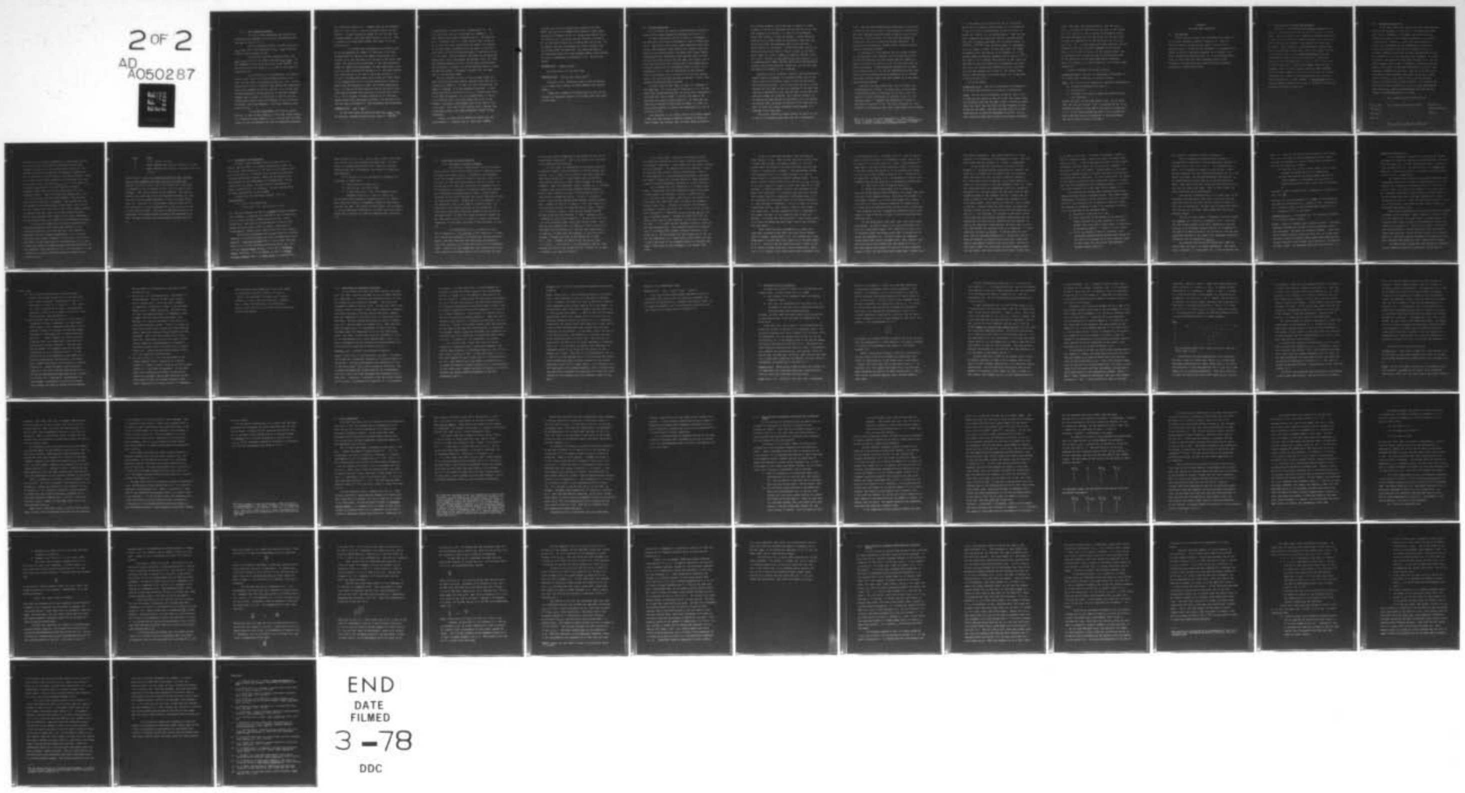


on which A* behaves in the same way as on a hollow rectangle R. If R contains no x's, and A* starts at the upper left corner of R, A* must eventually accept R, say without making more than m turns around R in either direction. If we choose the two spirals in S to have more than m turns, A* must thus accept S without ever reaching its ends, so that it accepts S even if there are x's at its ends, contradiction.

By Propositions 4.4.4, 6, and 7, it still makes no difference, in most cases, whether we define acceptance from a standard point such as the upper left corner, from all points, or from some point. We shall use the upper-left-corner definition from now on. The set $L_C(A)$ will be called the <u>language</u> of A, and will be denoted by $L(A)$.

AD-A050 287 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 9/4
SEQUENTIAL AND PARALLEL PICTURE ACCEPTORS.(U)
DEC 77 A ROSENFELD AFOSR-77-3271
UNCLASSIFIED TR-613 AFOSR-TR-78-0119 NL

2 OF 2
AD
A050287



END
DATE
FILMED
3 -78
DDC

4.4.3. The language hierarchy

We use the same terminology and notation for connected array languages as we did for rectangular array languages in Section 4.3.2.

As for rectangular arrays, we have $L_{DT}=L_T$ and $L_{FS} \underset{\neq}{\supset} L_{DFS}$, while the properness of $L_{TB} \supseteq L_{DTB}$ is an open question (and similarly for #P).

We can still show for connected arrays that $L_{D\#P} = L_{DTB}$; but it is no longer true that $L_{DFS}=L_{DTBFS}$. In the remainder of this section we prove these results. The proof of the first is based on [6]; that of the second is taken from [10].

As in Section 4.3.2, the behavior of a D#PA, A, while on the # part of its page becomes periodic after at most $|Q|$ time steps, with period at most $|Q|$. Let ρ be the string of U's, D's, L's and R's that defines the nonperiodic part of A's movement on the #s, and let σ be a string that similarly defines a single period of A's periodic movement. Suppose that A leaves Σ at the point P, and let B be the border along which Σ meets the component of #s into which A has moved. If A ever reenters Σ, this must happen at some point P' of B.

If the net displacement of A over an entire period σ is zero, A must reenter Σ (if at all) either during ρ or during the first period σ, i.e., in less than $2|Q|$ time steps, so that the behavior of A can be simulated internally by a DTBA that remains on Σ. Suppose that the net displacement is $(r,s) \neq (0,0)$. Then if A reenters during the kth period σ, k cannot exceed the length $|B|$ of B by more than a bounded amount, since the reentry point P' is on B. Thus A reenters in at most about $|Q||B|$ time steps, if it reenters at all.

It follows that the behavior of A while on the #s can be simulated by a DTBA, A', that uses B as a tape. In particular, A' can use B to compute the position of A relative to P, and the state of A, at each of the $|Q||B|$ time steps. For each of these positions, A' can also follow the border B, compute the displacement of each border point relative to P (see the proof of Theorem 4.4.5), and check whether this displacement is the same as the given position of A. If so, A' can mark that border point, erase all other traces of its computations on B, go to the marked point, erase the mark, enter the appropriate state, and resume simulating A on Σ. If none of the positions of A during the first $|Q||B|$ time steps coincides with a point of B, A will never return to Σ. [If A accepts Σ without returning to it, this must happen within $2|Q|$ time steps, so that A' can ascertain this fact as soon as A leaves Σ.] This discussion has established

<u>Theorem 4.4.8</u>. $L_{DTB} = L_{D\#P}$.//

On the other hand, we shall now show that $L_{DTBFS} \subsetneq L_{DFS}$. To this end, consider the set of thin (width 1), upright

L-shaped arrays of (say) x's on a background of #s. We first show that this set is accepted by a DTBFSA, A. In fact, A moves left until it bounces off a #, then moves up until it bounces off a #. A then verifies that it has #s on its left and right, moves down, and repeats the process. If, at some stage, A finds an x on its right, it verifies that there are #s on its left and below it, and moves to the x on its right. Here A verifies that there are #s above and below, moves right, and repeats the process, until its rightward movement bounces off a #. If all these verifications are successful, A accepts; if any verification fails, A stops and does not accept. Evidently A accepts iff. its input array is as described above.

We next consider the set of such L-shaped arrays for which the arms have equal length, and show that this set is accepted by a DFSA, A'. In fact, A' first verifies that its input is L-shaped, as in the preceding paragraph. It then moves "diagonally" across the #s, i.e., it moves alternately to the left and upward. If, on an upward move, A' hits an x, it verifies this x is at the upper end of the L, and accepts. Evidently, this happens only if the arms of the L are equal. Note that if the vertical arm is shorter than the horizontal arm, A' will move diagonally forever without accepting.

Finally, we show that no DTBFSA can accept just the equal-armed L's. Suppose that A" were such a DTBFSA.

Consider the set of one-dimensional arrays of the form $a^m b a^n$, and let A* be a DTBFSA that behaves on these arrays exactly as A" behaves on the L's, except that when A* is to the left of the b, its moves are 90° rotations of those of A" (so that when A" moves up, A* moves left, and so on). If A" accepts just the equal-armed L's, then A* accepts just the strings $a^m b a^n$ for which m=n. We could thus define a one-dimensional FSA, A**, that accepted exactly these strings; but this is impossible (see Section 3.3.2). We have thus proved

<u>Theorem 4.4.9</u>. $L_{DTBFS} \subsetneq L_{DFS}$.//

As in Section 4.3.2, we still have

<u>Theorem 4.4.10</u>. $L_{(D)T} \supsetneq L_{TB}$; $L_{DTB} \supsetneq L_{DFS}$.//

The proof is the essentially same as that of Theorem 4.3.7, using n-by-1 arrays, on which DTBFSA's can simulate DFSA's.

"Three-way" acceptors (see Section 4.3.3) will not be studied for connected arrays; it is evident that they would be very weak.

4.5 Closure properties

In this section we consider two-dimensional generalizations of the closure properties discussed in Section 3.5.

A finite intersection of TLs or TBLs is a language of the same type; the proof is analogous to that of Proposition 3.5.1. Note that in this proof, after completing each simulation, the automaton A must restore the original array and return to its starting point (the upper left corner) before beginning the next simulation; this requires that A be able to scan the non-#s and to find the upper left corner, and we have established that TLs and TBLs have these capabilities. A finite intersection of rectangular FSLs is an FSL, as in Proposition 3.5.1, since no restoration of the original array is necessary, and it is trivial to find the starting point. We leave open the question of whether a finite intersection of connected FSLs is an FSL.

A finite union of TLs, TBLs, or FSLs is a language of the same type. As in the case of Proposition 3.5.2, we can prove this nondeterministically, or we can give a deterministic proof for (TLs or) TBLs: A scans Σ and updates the positions and state of each simulation. We leave open the question of whether a finite union of DFSLs is a DFSL, as well as the question of whether the complement of an FSL of an FSL is an FSL.

Any singleton $\{\Sigma\}$ is a DFSL, since we can always design a DFSA that goes through a specific sequence of moves and state changes and verifies that its input array is exactly Σ.

By a similar argument, any finite set of arrays is a DFSL. A subset of an FSL is not necessarily an FSL; we can use the same example as in Section 3.5. The set of rectangular arrays that contain, or fail to contain, a given connected subarray Σ_0 is a DFSL, since we can design a DFSA that systematically scans the rectangle, and at each position, checks whether Σ_0 is present. On the other hand, the set of connected arrays that fail to contain a given Σ_0 (or even a given symbol x) is not a DTBFSL, as shown at the end of Section 4.4.2. The set of connected arrays that contain a Σ_0 is a nondeterministic TBFSL (the FSA moves around nondeterministically, and at each position, checks for Σ_0); we leave open the question of whether it is a D(TB)FSL.

Reversals, cyclic closures, closures, and concatenations cannot readily be defined for connected arrays, but we can define them for rectangular arrays in both the horizontal and vertical directions. It is clear that the row or column reversal of a TL, TBL, or FSL is a language of the same type, just as in Proposition 3.5.5, and the same is true when both rows and columns are reversed. By a similar argument, we can easily see that the set of reflections (in a horizontal or vertical axis, or both), or the set of 90° rotations (or 180°, or 270°), of a given connected array language is a language of the same type.

The cyclic closure or column closure (or both) of a TL or TBL is a language of the same type, as in Proposition

3.5.6. Here the TBA nondeterministically marks a row and/or a column of the given array Σ, and simulates an acceptor for the given language using the marks to define a cyclic shift. Similarly, the TA dissects Σ at the marks and creates an "unshifted" copy of Σ on which to do the simulation. We leave open the question of whether cyclic closures of rectangular array FSLs are FSLs.

In defining the concatenative row (or column) closure of a rectangular array language, or the concatenation of a set of such languages, we must assume that the numbers of rows (or columns) all match, so that the resultant array is still rectangular. These operations preserve TLs or TBLs, as in Propositions 3.5.7-8; we can nondeterministically segment a given rectangular array by marking a set of columns (or rows), and then check that each segment is in the appropriate given language.

The FSL's are not closed under (row or column) concatenation or closure*. To prove this, we first make an observation about the behavior of an FSA, A, with respect to a given (horizontal or vertical) slice of a rectangular array Σ. [Compare the discussion of blocks in the proof of Theorem 4.3.4.] Let S be a subarray of Σ consisting of m consecutive rows (say), where each row has length

*The proof in the following paragraphs is taken from K. Inoue, A. Nakamura, and I. Takanami, A note on two-dimensional finite automata, Information Processing Letters, to appear.

n. A can enter S at its top row at any of n positions and in any of q states, and can leave S at its bottom row at any of 2^n sets of positions and in any of 2^q sets of states (or n and q, if A is deterministic). Thus the number of mappings from (entering position, entering state) into (set of leaving positions, set of leaving states) is $(2^{n+q})^{nq} = 2^{(n+q)nq}$; note that this number is independent of m, the number of rows. On the other hand, the number of possible rows, say made up of two symbols 0,1, is 2^n, and the number of possible nonempty sets of rows is $2^{(2^n)}-1$. For large n, 2^n is greater than $(n+q)nq$, so that there are more sets of rows than there are I/O mappings, and there must exist two slices having different sets of rows that give the same mapping for a given A. (Of course, these sets of rows may have to be quite large, i.e., m may have to be as large as 2^n.)

We can now prove

<u>Proposition 4.5.1.</u> The set of rectangular FS languages is not closed under horizontal or vertical concatenation.

<u>Proof:</u> The set of all rectangular arrays on (say) 0 and 1 is obviously FS, and so is the set of all such arrays whose top and bottom rows are the same (A does a column by column scan, and checks that the first and last elements of each column are the same). We shall now show that the vertical concatenation of these two languages is not FS. [An exactly analogous proof can be used for horizontal concatena-

tion.] Note that this concatenation is just the set of arrays L in which the bottom row is the same as some nontop row. For any FSA, A, let S_1 and S_2 be rectangular arrays that have different sets of rows R_1 and R_2, but that yield the same I/O mappings for A, and let R be a row in R_1 but not in R_2. Let $\Sigma_1(\Sigma_2)$ be $S_1(S_2)$ with some arbitrary row ($\neq$R) appended at the top, and with R appended at the bottom. Thus Σ_1 is in L but Σ_2 is not. We can assume without loss of generality that A starts at the top row and moves down to the bottom row of its input array before accepting. Thus if A accepts Σ_1, it also accepts Σ_2, so A cannot accept exactly L.//

A similar argument can be used to prove

Proposition 4.5.2. The set of rectangular FS languages is not closed under horizontal or vertical closure.

Proof: Let Σ be an array obtained by vertically concatenating

a) An arbitrary array of 0's and 1's

b) A row of 2's

c) An array of 0's and 1's whose top and bottom rows are identical.

Clearly the set L of all such arrays is FS. On the other hand, the vertical closure L^+ of L is readily not FS, by an argument analogous to that used in the proof of Proposition 4.5.1: in the zones between the rows of 2's, we must now verify that the top row is identical to some non-bottom row, which cannot be done by an FSA.//

CHAPTER 5

CELLULAR ARRAY ACCEPTORS

5.1 Introduction

Cellular array acceptors are two-dimensional analogs of the cellular acceptors studied in Section 3.4. They are arrays of cells, all but finitely many of which are in a special state #. The non-#s may constitute a rectangular subarray, or merely a connected subarray; as in Chapter 4, we shall consider both possibilities. As in the one-dimensional case, we shall show that these acceptors define no new classes of langauges, but that they often make acceptance possible in a much shorter time.

5.2 Cellular array automata and acceptors

Informally, a two-dimensional cellular automaton is an array of "cells" each of which, at any given time, is in some state. The cells operate in a sequence of discrete time steps, at each of which every cell reads the states of its four horizontal and vertical neighbors, and changes to a (possibly) new state. Thus, formally, a two-dimensional cellular automaton K is defined by specifying a pair (Q,δ), where Q is the set of states, and δ is the transition function that maps quintuples of states into sets of states (or into single states, if K is deterministic) -- i.e., $\delta: Q^5 \to 2^Q$ (or $\to Q$). For each cell c of K, δ maps the quintuple (state of left neighbor, state of right neighbor, state of upper neighbor, state of lower neighbor, state of c) into the set of possible new states of c, where in the deterministic case, this set always consists of a single element. A configuration of K is simply a mapping from IxI (the pairs of integers) into Q which specifies the state of each $c \in K$.

5.2.1 Neighborhood size [11]

In the above definition we used only the four horizontal and vertical neighbors of each cell c. We could have used other sets of neighbors; for example, we could have also allowed the states of the diagonal neighbors of c to affect c's next state. However, a transition that depends on a larger set of neighbors can be simulated by a sequence of transitions involving only the four neighbors, so that a cellular automaton K that uses a larger neighborhood set can be simulated by a K' that uses only four neighbors, at a rate of several time steps of K' for each time step of K. For example, suppose that we want to simulate a transition that depends on the states of the eight (horizontal, vertical, and diagonal) neighbors of each cell. One way to do this is for each cell, at even-numbered time steps, to go into a state of the form $(q_1,\ldots,q_5)$, where the q's are the previous states of the cell and its four neighbors. If we denote the state of the cell at position (i,j) at a given odd-numbered time step by q_{ij}, then at the following time step the cell states in the neighborhood of (i,j) are

$$\begin{array}{ccc}
 & (q_{i-1,j+1}, q_{i+1,j+1}, q_{i,j+2}, q_{ij}, q_{i,j+1}) & \\
\begin{array}{l}(q_{i-2,j}, q_{ij},\\ q_{i-1,j+1},\\ q_{i-1,j-1},\\ q_{i-1,j})\end{array} & (q_{i-1,j}, q_{i+1,j}, q_{i,j+1}, q_{i,j-1}, q_{ij}) & \begin{array}{l}(q_{ij}, q_{i+2,j},\\ q_{i+1,j+1}, q_{i+1,j-1},\\ q_{i+1,j})\end{array} \\
 & (q_{i-1,j-1}, q_{i+1,j-1}, q_{ij}, q_{i,j-2}, q_{i,j-1}) &
\end{array}$$

The cell at (i,j) now has available to it the states (at the previous step) of itself and its horizontal and vertical neighbors, stored in its own current state; and it also has available the previous states of its diagonal neighbors (as well as the states of other cells at city block distance 2 from it), stored in the current states of its horizontal and vertical neighbors (e.g., the state $q_{i-1,j+1}$ of (i,j)'s upper-left neighbor is available as the first term of its upper neighbor's state, and also as the third term of its left neighbor's state). Thus at the next odd numbered step, cell (i,j) can go into a new state that depends on the previous states of all eight of its neighbors. This process can then be repeated at succeeding pairs of time steps. Note that it not only requires two time steps of the four-neighbor automaton to simulate each time step of an eight-neighbor automaton, but it also requires a much larger set of states (quintuples of the original states are needed at even-numbered time steps).

It can also be shown that a cellular automaton whose transition function involves only <u>two</u> neighbors in non-collinear directions, say north and east, can simulate an automaton that has a larger neighborhood -- e.g., the four horizontal and vertical neighbors. To see this, let us have each cell, at time steps that are not multiples of 4, go into a state that contains information about the previous states of its neighbors. Thus the cell at (i,j), at time steps 1, 2, and 3 (modulo 4), contains information about the states of the following cells:

Step	Cells
1	(i,j), (i+1,j), (i,j+1)
2	These together with (i+2,j), (i+1,j+1), (i,j+2)
3	These together with (i+3,j), (i+2,j+1), (i+1,j+2), (i,j+3)

Hence at step 3, cell (i,j) knows the states of cell (i+1,j+1) and its four neighbors (i+2,j+1), (i,j+1), (i+1,j+2), and (i+1,j), so that at step 4, cell (i,j) can go into a new state that depends on the old states of cell (i+1,j+1) and its four neighbors. The entire process is then repeated. Note that it requires four time steps of the two-neighbor automaton per time step of the four-neighbor automaton, as well as a much larger state set. Moreover, the two-neighbor simulation does not remain stationary; it shifts downward and to the left by one unit in each four time steps. This type of simulation would not be appropriate for bounded cellular automata (see immediately below). We shall consider only four-neighbor automata from now on.

5.2.2 Boundedness and acceptance

We shall assume henceforth that the state set Q is finite, and that there is a special state $\# \in Q$ such that, in the initial configuration of K, all but a finite number of the cells have state #. In Section 3.4 we further required that the non-# cells form a string. There are two possible generalizations of this requirement to two dimensions: We can require that the array Σ of non-# cells be rectangular in shape, or merely that Σ be connected. We will consider both of these possibilities in this chapter.

We say that the transition function δ of K is #-preserving if

$$\# \in \delta(q,r,s,t,u) \text{ implies } u=\#$$
$$\delta(q,r,s,t,\#) = \{\#\} \text{ for all } q,r,s,t \text{ in } Q$$

If δ has this property, we call K a bounded cellular automaton. [As in Chapter 3, inability to create #s is not a restriction on K; we shall therefore assume from now on that δ never creates #s.] Note that in the bounded case we may as well assume that the array of cells is finite, consisting of Σ surrounded by a border of #s, since the #s will never change.

The notion of acceptance of an input array by a cellular automaton K is defined exactly as in the one-dimensional case. Formally, a cellular acceptor (CA), C, is a triple (K,Q_I,Q_A), where K is a cellular automaton with state set Q, $Q_I \subseteq Q$ is a set of initial states, and $Q_A \subseteq Q$ is a set of accepting states, with $\# \in Q_I$. If K is bounded, we call C a bounded cellular acceptor (BCA). An input array is a configuration whose states are all in Q_I. We say that C accepts this array (or, for brevity, that C accepts the non-# part Σ of this array) if repeated application of K's transition function, starting from this configuration, can lead to an "accepting configuration".

As in Section 3.4.1, we can define an accepting configuration in three ways:

a) Every $c \in C$ has its state in Q_A.

b) Some $c \in C$ has its state in Q_A.

c) A particular $c_0 \in C$ -- e.g., the leftmost of the uppermost cells of Σ -- has its state in Q_A.

We can show, using analogs of Propositions 3.4.1-4, that in most cases the classes of languages accepted in these three ways are the same. The proofs are much simpler for rectangular Σs than for arbitrary connected Σs; we shall treat the rectangular case in Section 5.3, and the connected case in Section 5.4.

5.3 Rectangular cellular acceptors

5.3.1 Rectangular cellular languages

The analogs of Propositions 3.4.2-4 for rectangular cellular arrays follow readily from the observation that the upper left corner cell c_0 ($\equiv$ leftmost of the uppermost cells) can uniquely identify itself at the start, since it is the only cell that has #s both above it and to its left. In Proposition 3.4.2, when any cell enters an accepting state in the simulation, it initiates an accepting signal that spreads in all directions (through non-#s); since Σ always remains connected (#s are never created), this signal eventually reaches c_0, which then accepts. Proposition 3.4.3 is immediate: in the simulation, only c_0 can enter an accepting state. In Proposition 3.4.4, when c_0 accepts in the simulation, it initiates an accepting signal that spreads in all directions (through non-#s), so that this signal eventually reaches all of Σ and causes every cell to accept; by using a half-speed simulation, we can assure this even if the simulation is growing.

For Proposition 3.4.1, c_0 identifies itself, and C' simulates C. If c_0 accepts in the simulation, it sends out a signal that spreads in all directions through accepting states. We shall show next how this signal can generate a reply that reaches c_0 iff. every non-# cell is in an accepting state. [Here we use the assumptions that accepting states are never rewritten and cannot cause #s to be rewritten; we shall see later how these assumptions can be avoided at the cost of a much slower simulation.] Thus c_0 can accept iff. every cell of the simulation has accepted.

If C is a BCA, the propagation of the reply signal can be done very simply, since the array Σ of non-#s remains rectangular. Let c_1 and c_2 be the cells in the upper right and lower right corners of Σ, respectively; they are uniquely identified by having #s as their right and upper (or lower) neighbors. When the acceptance signal α reaches c_2, it initiates a signal β that can only spread leftward through accepting states; thus β reaches all cells of the bottom row of Σ iff. they are all in accepting states. In turn, β initiates a signal γ that can only spread upward through accepting states; thus γ reaches the top of a column of Σ iff. that column consists entirely of cells in accepting states. When γ reaches c_1, it initiates a signal δ that can only spread leftward through cells that have received γ. Thus δ reaches c_0 iff. γ reached the top of every column of Σ, so that every cell of Σ is in an accepting state, and c_0 can finally accept. Note that the time required for α to reach c_2 is $h+w-2$, where h is the height and w the width of Σ; and the time required for δ to reach c_0, if every cell has accepted, is also only $h+w-2$. (Indeed, β reaches the cells of the bottom row at times $0,1,\ldots,w-1$; γ reaches the cells of the top row at times $h-1, 1+(h-1),\ldots,(w-1)+(h-1)$; δ is initiated at time $h-1$, and reaches each cell of the top row just as γ reaches it, so that it reaches c_0 at time $(w-1)+(h-1)$.)

If C is not a BCA, Σ may not be rectangular, but we can handle the propagation of the reply signal by generating a rectangle R containing Σ, and transmitting the signals throughout that rectangle. [In Section 5.4 we will describe a method of signaling that does not require going outside Σ even if Σ is nonrectangular.] In the following paragraphs we describe one method of doing this, derived from [12], which generates a square (oriented diagonally) centered at c_0 that contains Σ.

Let us assume that the simulation of C is done at half speed. Concurrently with this simulation (using first terms of state pairs), we propagate a single α outward from c_0; i.e., any cell, say in state q (non-# or #), not having α as first term, that sees first term α as one of its neighbors becomes a pair (α,q). Moreover, we record the direction(s) from which α reaches each cell; e.g., we can do this by using eight versions of α, denoted by $\alpha_N, \alpha_S, \alpha_E, \alpha_W, \alpha_{NE}, \alpha_{NW}, \alpha_{SE}$, and α_{SW}, to indicate whether α reached the given cell from its north, south, east or west neighbor, or from both its north and east, ..., or south and west neighbors (readily, these are the only possibilities). This process generates an expanding wave of α's shaped like a square oriented diagonally, with center at c_0. We will assume that the propagation of the α's, like the simulation of C, takes place at half speed; thus the α's add a new layer to the expanding square at alternate time steps. (The reason for this assumption will become clear later.)

As the α's reach each new layer, they send back an "echo" signal β to c_0. (This is done once only at each layer; to insure this, we mark the newely created α's, say with primes, which are erased at the next time step, and only the primed α's initiate the β signals.) The directional information contained in the α's is used to direct β so that it travels back to c_0 along the shortest possible paths. For example, if cell c, say in state (α,q), has an α'_N, α'_{NE}, or α'_{NW}, or an $(\alpha_N,\beta), (\alpha_{NE},\beta)$ or (α_{NW},β), in the first term of its south neighbor, it goes into state $((\alpha,\beta),q)$; and similarly for the other three directions. It is easily seen that the β signals from a given layer of α's generate a contracting wave of β's that occupies the successive layers of the square centered at c_0, from outermost to innermost. If the β's start at a layer t steps away from c_0, they reach c_0 (from all four sides) t time steps later. We shall assume that the β's propagate at full speed, and that β's are erased as soon as they are created. It follows that the successive α layers create waves of β's that are spaced two units apart, and do not interfere with each other.

Each β wave can carry information to c_0 about the α layer that initiated it and about the other layers through which it passed. Specifically, if a β is initiated by a cell whose state is of the form (α,q), where $q \neq \#$, we shall denote it by β^*. (A given β may be initiated by two α cells; we use β^* if either of these cells has a non-# second term.) Moreover, if a β reaches a cell whose second term is neither # nor an accepting state of C, we change it to β^*. When β's reach a cell c from two or more of its neighbors at once, then if any of these β's is a β^*, we use β^* at c also. These conditions imply that if any cell in the initiating α layer is non-#, or if any cell within the square of α's is neither # nor accepting, then the β that reaches c_0 will be a β^*. Conversely, if an unstarred β reaches c_0 (i.e., it receives β's from all four neighbors, and its own state is # or accepting), we know that

a) Every cell in the initiating layer is #

b) Every cell in the square is # or accepting.

Since the non-#s always remain connected, (a-b) imply that every non-# cell is an accepting state. Thus when this happens, c_0 can accept. Moreover, if it ever does happen, this process will detect it, since accepting states do not change and do not cause #s to be rewritten; thus if every non-# is accepting, Σ stops growing, and the α propagation catches up with it and reaches a layer of all #s, so that a β wave is generated that satisfies (a-b).

In the construction just given, the α wave never stops propagating. In order to stop it, c_0 (upon acceptance) can send out a full-speed signal γ that erases the α's and β's as it encounters them, and erases itself when it reaches the #s beyond the outermost α's. If this is done, the expanding square never grows beyond a bounded size. Note, however, that this size may be much greater than the radius of Σ; e.g., Σ may stop growing long before its cells all go into accepting states, so that the square gets much bigger than Σ before the β

wave detects acceptance. Thus after every cell of Σ has accepted, the propagation of the acceptance signal β may take much longer than the diameter of Σ, which is undesirable.

We can reduce the time to be on the order of the diameter of Σ by modifying the construction as follows: The square of α's expands (at half speed) until its outermost layer consists of #s only, so that it entirely contains Σ. The β wave relays this information to c_0, which sends out a γ signal (at full speed) to stop the expansion of the α's. [Here the β wave must carry two independent pieces of information, one about #s in the outer layer, and the other about #s and accepting states in the interior; and the γ signal does not erase α's and β's, but only "freezes" the outermost α's so they cannot expand. Note that by the time the β wave reaches c_0 and the γ signal reaches the outer layer, the square has tripled in size; but its radius is still of the same order as that of Σ.] The outermost α's continue to send back β waves even though they have stopped expanding (these waves are now consecutive, but still do not intefere with one another), so that c_0 can continue to check whether the outer layer is still all # and whether the interior is all # or accepting. If these conditions are met, c_0 accepts. If non-#s reach the outer layer (due to continued expansion of Σ) before the cells of Σ have all accepted, c_0 discovers this from the β signal and sends a full-speed δ signal to the outer layer. Arrival of this signal causes the outer layer to start expanding again (at half speed), and also to send a full-speed ε signal back to c_0; when c_0 receives ϵ, it sends a full-speed γ signal to stop the expansion again. During this time, the square again triples in size. If the non-#s again overtake (or have already overtaken) its outer layer, the process can be repeated. If the cells of Σ ever all accept, Σ stops growing, so the square eventually expands beyond Σ, at which time the β wave detects the conditions for acceptance. In this construction, the radius of the square is never more than triple the radius of Σ; hence the time for propagation of the acceptance signal Σ, after all cells have accepted, is on the order of the diameter of Σ.

Up to now we have assumed that accepting states never change and never cause # to become non-#s. Even without these assumptions, we can still verify that every cell (of a simulation of C) has accepted, but the process is now much slower. The approach is analogous to that in the appendix to Section 4.3.1; it consists of the following steps:

a) We simulate C one step at a time.

b) After each step of the simulation, we construct a square centered at c_0 that contains Σ, and check whether ever non-# cell in this square is simulating an accepting state of Σ. [Here C' is not a BCA!]

c) If so, c_0 accepts; if not, c_0 initiates a two-dimensional synchronization process that causes all the cells of the square to go into a special state, at which point the non-# cells simultaneously simulate another step of Σ.

This process is repeated as often as necessary.

The two-dimensional synchronization process is a straightforward extension of the one-dimensional process described earlier; see, e.g., [13]. To illustrate how it works, let us suppose for simplicity that we have a w-by-h upright rectangle R of cells with distinguished cell c_0 in the upper left corner. By the one-dimensional construction, c_0 can synchronize the left column of R, and this takes O(h) steps. Each cell in the left column can then synchronize its row, and this takes O(w) steps. Thus in O(h+w) steps, the entire rectangle can be synchronized.

The analogs of Propositions 3.4.1-4 shows that the definitions of acceptance by some cells, all cells, and the upper-left cell (or any cell that can uniquely identify itself) are all equivalent. We shall use the upper-left definition from now on, and shall always assume that cell c_0 is uniquely marked.

The set of input arrays Σ accepted by C will be called the <u>language</u> of C, and will be denoted by $L(C)$. Note that $L(C)$ can contain arrays of any size, depending on how many cells of C are initially in non-# states. [C is regarded as the same CA no matter how big Σ is, as long as C has the same transition function.] The class of all languages accepted by (D)(B)CA's will be denoted by $L_{(D)(B)C}$.

We conclude this section by showing how a DBCA can verify that its input Σ is indeed rectangular. Note first that a connected Σ is rectangular iff. it has no concave corners, i.e., there are no 2-by-2 patterns exactly one of whose cells is # (this will be proved immediately below). Next, note that the presence of concave corners can be detected, in only two time steps, as follows:

> Any cell that has a # as a north, south, east, or west neighbor is marked N,S,E, or W, respectively.
>
> Any cell that has an S as east neighbor and an E as south neighbor, or analogously for the pairs of directions S and W, N and E, and N and W, is marked K.

Evidently a cell is marked K iff. it belongs to a concave corner, e.g., $\begin{matrix}K & S\\ E & \#\end{matrix}$.

Finally, cell c_0 can determine, in a number of time steps on the order of the diameter of Σ, whether or not Σ contains any K's; a method of doing so will be described in Section 5.4. If Σ contains no K's, c_0 can accept.

<u>Proposition 5.3.1</u>. A connected set Σ is an upright rectangle iff. it has no concave corners.

<u>Proof</u>: Clearly a rectangle has no concave corners. Conversely, consider a maximal set of runs in which consecutive rows of the array meet Σ and which are pairwise adjacent. Since there are no concave corners, these runs must line up at both ends, and none of them can merge or split; hence they constitute an upright rectangle, and no other run of points of Σ is adjacent to them. Since Σ is connected, they must thus be all of Σ.//

Appendix to Section 5.3.1

In this appendix we show how to construct the upright rectangle $R(\Sigma)$ that just contains a given connected Σ (so that there are points of Σ in the top and bottom rows, and left and right columns, of $R(\Sigma)$). We will call $R(\Sigma)$ the framing rectangle of Σ, and we will denote ${}^{\Sigma}$ its extremal rows and columns by t,b,ℓ, and r, respectively.

Our construction is based on a propagation process that repeatedly fills concave corners; in other words, if a # cell has two non-# neighbors on adjacent sides of it (N and E, N and W, S and E, or S and W), we change it to a non-#, say to ♮ . We shall show that if this process is iterated until no further change takes place, the ♮s together with Σ itself constitute exactly $R(\Sigma)$. Moreover, the process stops in a number of time steps at most equal to $h+w-2$, where h and w are the height and width of $R(\Sigma)$.

Note first that when we fill concave corners, the framing rectangle is not enlarged. Indeed, clearly a # above t, below b, to the right of r, or to the left of ℓ cannot belong to a concave corner; hence no step of the propagation process can create a ♮ outside $R(\Sigma)$. Since $R(\Sigma)$ is finite, the process must stop eventually. On the other hand, when the process stops, the resulting $\overline{\Sigma}$ has no concave corners, so by Proposition 5.3.1 must be an upright rectangle; and clearly the only such rectangle that meets t,b,ℓ and r must be $R(\Sigma)$ itself.

It remains only to show that this process of constructing $R(\Sigma)$ takes at most $h+w-2$ time steps. We do this in several steps:

a) Let us first observe that $R(\Sigma)$ cannot contain a 4-path ρ of #s that has one end on t and the other on b. This is because Σ touches both ℓ and r, so that there is a path ρ' of non-#s in $\Sigma \subseteq R(\Sigma)$ from ℓ to r, and ρ' would have to cross ρ, contradiction. Similarly, there can be no 4-path of #s in $R(\Sigma)$ that has one end on ℓ and the other on r.

b) Next, we show that for any point $P \in R(\Sigma)$, there is a quadrant in which Σ surrounds P; in other words, there exists a pair of adjacent directions (N and E, N and W, S and E, or S and W) such that any 4-path starting at P that moves only in these directions must hit Σ. Indeed, suppose not; then in all four quadrants there are 4-paths from P that reach the border of $R(\Sigma)$ without meeting Σ. The path ρ_1 in the NE quadrant, for example, must hit the border either on t or on r; suppose it hits t. If the path in the SE or SW quadrant hit the border on b, then by concatenating this path with ρ_1 we would get a 4-path of #s from t to b (through P), contradicting (a). Hence the path in the SE quadrant must hit r, and that in the SW quadrant must hit ℓ, so that by concatenating these paths we get a 4-path of #s from ℓ to r (through P), contradicting (a). The argument is analogous if ρ_1 hits r; we would then consider the paths in the NW and SW quadrants.

Thus we obtain a contradiction in any case, so that (b) must be true.

c) Let $P \in R(\Sigma)$, $\notin \Sigma$, and suppose that Σ surrounds P in the NE quadrant. Thus if we move (e.g.) upward from P, we must hit Σ; if we then move rightward (if possible), we must hit Σ again; if we then move upward, we hit Σ again; and so on. The path of #s (or ♮s) along which we move is a 4-geodesic, since it makes left and right turns alternately (see Section 2.9); thus the nth point on the path is at city block distance n from P. Since Σ is finite, we cannot get farther than a finite city block distance from P; hence this process of path construction must terminate. But it can only terminate at a point which has Σ both above it and on its right, so that neither upward nor rightward motion is possible; thus the terminal point of the path is at a concave corner of Σ. Since the path always remains inside $R(\Sigma)$, its length is less than $h+w-2$.

d) For any $P \in R(\Sigma)$, $\notin \Sigma$, let k be the length of the longest geodesic from P through #s to a concave corner of Σ; thus $k < h+w-2$. When we fill the concave corners of Σ, this path is shortened. On the other hand, (a-c) evidently still hold, so that we can repeat this argument, and at each iteration of concave corner filling, the longest geodesic length from P through #s to a concave corner of Σ decreases.

Since initially this length was $< h+w-2$, the number of iterations required to reduce it to zero is $< h+w-2$. At this stage, P itself is at a concave corner of Σ, and at the next iteration Σ swallows up P. Since P was arbitrary, this means that Σ grows to fill all of $R(\Sigma)$ in at most $h+w-2$ iterations, as was to be proved.

5.3.2 Equivalence to sequential acceptors

In this section we prove that TAS's and CA's can simulate each other, and that the same is true for TBA's and BCA's. This shows that, in introducing two-dimensional CA's, we have not created any new classes of languages. The proofs are analogous to those in Section 3.4.2. To show that T(B)A's can simulate (B)CA's, we need to prove that a DT(B)A can systematically scan its array of non-#s, and at each point, compute what the transition of the (B)CA's cell at that point would have been. This has already been shown for DTA's in Proposition 4.4.1, and for DTBA's in Theorem 4.4.2. Note that even when the input array Σ is rectangular, we still need the traversal result of Proposition 4.4.1 for the TA/CA proof, since Σ may grow. On the other hand, for rectangular Σ's we do not need Theorem 4.4.2 in the TBA/BCA proof, since Σ remains rectangular, and its traversal is straightforward, e.g., row by row as in the proof of Proposition 4.2.1.

Theorem 5.3.2. (D)T(B)A's can simulate (D)(B)CA's.

Proof: Given a CA, C, with transition function δ, we define a TA, A, that has a vocabulary consisting of states and pairs of states of C. Any any given stage of the simulation of C by A, the non-# array Σ has a pair of states at each point, one of which represents the current state of the corresponding cell of C, while the other represents the preceding state of that cell. (Initially, Σ consists of just the initial states of C's cells.) To simulate one transition of C, A systematically scans Σ. At each point P of Σ, A also examines P's neighbors, records the current states of the corresponding cells of C, returns to P, and computes a new state of the cell at P using δ. A then erases the previous-state information at C (if any) and replaces it by new-state information, but leaves the current-state information intact (since it may be needed to compute new states for neighbors of P that have not yet been processed). A also marks P to indicate that it has been processed, and resumes scanning Σ. When the scan is complete, every point of Σ is marked, and contains a pair of states one of which is the former current state (now previous), and the other is the new state (now current), at that point. A now scans P again, erases all the marks, and returns to its starting point; it is now ready to simulate the next transition of C. A accepts iff. its simulation of C accepts.

If C is not a BCA, A must also visit all the #s adjacent to Σ during its scan, check their neighbors, and rewrite them as non-#s if C would have done so; thus if C is not a BCA, A cannot be a TBA. On the other hand, if C is a BCA, the simulator A can be a TBA, and if C is deterministic, so is A. Note that a very large number of transitions of A is needed to simulate a single transition of C; the simulation is very inefficient. (Compare the proof of Theorem 3.4.5, which provided the fastest possible simulation in the one-dimensional case.)//

<u>Theorem 5.3.3</u>. Cellular array acceptors simulate Turing array acceptors.

<u>Proof</u>: Given a TA, A, with transition function δ, we define a CA, C, that simulates A just as in the proof of Theorem 3.4.6. Initially, the upper left cell c_0 goes into a special state representing the initial state q_0 of A, the symbol α at that position, and a direction in which A can move when it is in state q_0 and reads symbol α. [This is easy for rectangular input arrays, where c_0 is unique, but it is not so easy for arbitrary connected arrays; see Section 5.4.] At subsequent transitions, the special state moves from cell to cell just as it did in the one-dimensional case, except that four move directions are now possible. At any step, exactly one cell is in the special state, corresponding to a position and state that A could have been in at that step. This process continues until the special cell's state represents an accepting state of A. At this point, the special cell propagages an acceptance signal to c_0, which can then accept, so that C accepts iff. A does. Note that if A is deterministic, so is C, and if A is tape-bounded, so is C (if we modify the simulation to handle the steps in which A bounces off #s without requiring # cells of C to change state). As in Theorem 3.4.6, this simulation is the fastest possible, since C requires only one time step to simulate each time step of A after the first (except for the propagation of the acceptance signal at the end).//

Theorem 5.3.2-3 immediately imply

Theorem 5.3.4. $L_{(D)C} = L_{(D)T}$; $L_{(D)BC} = L_{(D)TB}$.//

Note that we have proved these theorems not only for rectangular input arrays, but for arbitrary connected ones, except for the problem of c_0 initially identifying itself (in the proof of Theorem 5.3.3) in the arbitrary connected array case; this will be taken care of in Section 5.4.

5.4 Connected cellular acceptors

To extend the results of Section 5.3 to arbitrary connected arrays, we need to show that, in a DBCA,

a) The leftmost of the uppermost cells can uniquely identify itself.

b) This cell can interrogate all the cells (e.g., as to whether or not they are in accepting states) and know when they have finished replying.

We shall, in fact, show that these tasks can be carried out in time proportional to the perimeter or diameter of the given array.

Given that (a-b) can be done, it is straightforward to extend the results of Section 5.3 to connected arrays. The analogs of Propositions 3.4.2-4 require that (a) holds, while for Proposition 3.4.1 we require both (a) and (b) (in the non-BCA case, (b) is not needed, since we can use the expanding-square construction of Section 5.3.1). The slower procedure based on simulation one step at a time, in conjunction with scanning and synchronization, also applies in the connected case, provided that (a) holds. Note that we also need (b) to prove that a DBCA can tell whether its input array is rectangular, as pointed out in Section 5.3.1.

Theorem 5.4.1. There exists a DBCA in which the leftmost of the uppermost cells eventually goes into a special state, while no other cell ever goes into that state.

Proof (see [4]): Initially, each non-# cell c determines which of its neighbors (if any) is #, and then copies the corresponding information from its non-# neighbors. This enables c to determine, if it is on a border of Σ, how the border following algorithm BF (Section 2.6) would behave at it. In particular, c can tell which of its neighbors would precede or succeed it at each visit of BF to it. If c would be visited more than once by BF, it keeps the information about each visit separate by associating it (e.g., via a suitable subscript or position in a 4-tuple) with the particular # neighbor(s) that BF would examine at that visit. For example, if the neighborhood of c is

```
a # #
b c d
# # #
```

and borders are always followed by keeping the #s on the left, then the predecessor of c with respect to its upper # neighbor is a, while its predecessor with respect to its lower # neighbor is d.

Based on this local analysis, c can tell in particular, for each visit of BF to it, whether its predecessor lies higher or lower than it; in the former case, c associates a - (minus) with that visit, and in the latter case, a +. In addition, if c is an upper left corner (i.e., its left and upper neighbors are both #), it marks itself with U. All of this initial processing requires only a bounded number of time steps.

We now independently process the +'s, -'s, and U's on each border of Σ. As already pointed out, even if two borders pass through a given cell, their processes will not interfere, since the marks know which # neighbor(s) of c they are associated with. We can thus describe the processing for a single border B.

In this processing, the -'s shift forward along B (i.e., in the direction that BF would take), and the +'s shift backward (in the reverse direction). When a + and a - meet, they cancel out; and when a + or - hits a U, that U is erased. We claim that when this process has gone to completion, the +'s and -'s will all have cancelled, and all the U's will be erased <u>except for those on the highest row of B</u> (if any); this will be proved immediately below. If B is the outer border of Σ, there will exist such U's on the highest row of Σ. On the other hand, if B is a hole border, it is clear that there cannot exist U's on its highest row, so that in the case of a hole border, all the U's will be erased. Thus when we process all the borders of Σ in this way, the only remaining U's will be those on the highest row of Σ.

To prove that all U's not on the highest row of B will be erased, consider a U that is not a highest point of B; then as we move forward along B from U, we eventually reach a higher point. Let P be the first such point; then on the segment of B between U and P, there are more +'s than -'s. Now consider what happens as the +'s shift backward and the

-'s shift forward. If a - reaches U from its other side, U is erased; but if no such - reaches U, the + at P eventually reaches U (since only the -'s between U and P are available to cancel the +'s) and erases it. (Note that by definition, P must also be higher than any other U's between U and P, so that they too get erased.)

Conversely, let U be on the highest row of B; then as we move forward along B starting at U, the net number of +'s can never exceed the net number of -'s, so that no backward shifting + can ever reach U. Similarly, as we move backward along B starting at U, the net number of -'s (which are upward moves when we go backward!) can never exceed the net number of +'s, so that forward shifting -'s can never reach U. We can think of the +'s and -'s between any two consecutive highest U's (or all around B, if there is only one highest U) as constituting a well-formed parenthesis string, which cancels itself out completely in the course of the shifting; see (d) at the end of Section 3.4.3.

As described up to now, the cancellation process leaves the highest U's intact, but these cells can never know that they are highest U's, since no matter how long they wait, they can never know that a + or - from a very distant higher point is not still on its way toward them. In order to allow these U's to discover that they are highest, we must make a small modification in the cancellation process. When a + and a - meet, instead of cancelling, they turn into "ghosts", denoted by +' and -', which continue to shift in the same directions. When a +' and a -' meet, they simply pass each other; but when a +' meets a -, the +' is erased (without affecting the -), and when a -' meets a +, the -' is erased. It is easily seen that, on any segment of B bounded by two highest U's, at least one +' and -' (namely, the last ones to be created) will not be erased, and these will reach the U's. Thus when a U has been hit by both a +' and a -', it knows that no + or - can ever hit it, so that it must be a highest U. To illustrate this process, we show the successive shifting and cancellation steps for a simple example:

<u>Step</u>

```
 1        U1  -   -   +   -   +   -   -   +   +   +   U2
 2                -   -'  +'  -'      -   +   +
 3                    -   -'      -'  +'  +
 4                        -   -'  +'  +
 5                            -   +
 6                            +'  -'
```

(All succeeding steps, the +' moves left to U_1, and the -' moves right to U_2.)

Concurrently with and independently of all the above, any cell c that is a lower right corner (i.e., whose lower and right neighbors are #s) marks itself with L. By a process analogous to that described for the U's, the L's in the lowest row of B identify themselves. (Note that, like the U's, they can exist only on the outer border B_0 of Σ, not on hole borders.) When an L has identified itself, it sends out a signal α that travels clockwise around B_0 (i.e., with the #s on the left, starting with the # below the L). If α hits another L, it is erased. If it hits a U which has not yet been identified as highest, it waits until that U is either erased or found to be highest. In the former case, the signal continues to travel. In the latter case, that U must be the leftmost of the uppermost points of Σ, since it is the first of the highest U's that was reached by α starting from a lowest L. In other words, that U is the desired c_0.

The entire process of uniquely identifying c_0 takes time proportional to the perimeter of Σ. In fact, the initial steps take a bounded amount of time; the identifications of the highest U's and lowest L's take times that are fractions of the perimeter; and the identification of c_0 using α takes time less than the outer perimeter.//

In proving the analogs of Propositions 3.4.1-4 for non-BCA's, we can first identify c_0, e.g., by creating a square that contains Σ (see Section 5.3.1), then scanning the square row by row until the leftmost uppermost non-# is found. We can then synchronize the square and initiate the simulation; the proofs now proceed as in Section 5.3.1. The time required, as pointed out there, is proportional to the (greatest) diameter of Σ.

For BCA's, the simulation and the process of identifying c_0 can proceed concurrently. (We can do this for non-BCA's also, if we do the identification of c_0 on a copy of the original Σ, while allowing another copy of Σ to grow, if necessary, under the simulation.) In Proposition 3.4.1, when c_0 has identified itself and has also (later or already) accepted in the simulation, it sends out an interrogatory signal that generates a reply as in the theorem to be proved next (or, for non-BCA's, it can use the same method as in Section 5.3.1). In Proposition 3.4.2, any accepting cell sends out an acceptance signal; when it reaches c_0 (or when c_0 identifies itself and finds that this signal has already reached it), c_0 accepts. In Proposition 3.4.3, only c_0 can accept, and only after it has identified itself. In Proposition 3.4.4, when c_0 accepts (or identifies itself and finds it has accepted), it sends out an acceptance signal that causes every other cell to accept. The time required for transmission of these signals is at most the diameter of Σ; as shown in Section 2.8, this is less than the perimeter of Σ.

We conclude this section by proving

<u>Theorem 5.4.2</u>. There exists a DBCA with a distinguished cell c_0 such that c_0 goes into a special state after it has sent out a signal and received a reply from the cell farthest from it.

<u>Proof</u>: If c_0 is the upper left cell of a rectangular array, this is easy: c_0 sends out its signal, which propagates from cell to cell; i.e., each cell copies the signal from its

neighbors. The lower right cell is uniquely identified by having #s below it and on its right, and this is the cell farthest from c_0. When this cell receives the signal, it emits a reply, which propagates from cell to cell until it reaches c_0. When c_0 receives the reply, it knows that the farthest cell has received its signal and replied, and can go into the special state.

For connected arrays, we could use the fact that DBCA's can simulate DTBA's, as was proved in Section 5.3.2. Cell c_0 could thus initiate a DTBA simulation that systematically scanned the array (see Theorems 4.4.2 and 4.4.5), and when finished, erased all its marks and returned to c_0; at this point, c_0 could accept. However, this approach is very slow (compare Section 5.5). Instead, we give a construction that makes extensive use of the DBCA's parallelism, and requires only time proportional to the intrinsic diameter of the array.

First, c_0 sends out its signal s, which propagates from cell to cell. If a cell c receives s from exactly one of its neighbors, c stores the direction of that neighbor (as well as copying s). If c receives s from two or more neighbors, it stores only one of their directions, according to an arbitrary preference ordering (e.g., L,R,U,D). Note that c receives s after a number of time steps equal to c's city block distance from c_0.

When a cell c receives s from all of its (non-#) neighbors, so that it has no neighbors that have not yet received s, the distance of c from c_0 must be a local maximum. Such c's initiate a reply signal r, which will propagate back to c_0 along the paths indicated by the directions stored in the cells; in other words, if c has received r, its neighbor c' accepts r from it only if c' is the neighbor of c in the direction that c had stored. Moreover, if c' has two (or more) neighbors, say c_1, c_2, for which it is the stored-direction neighbor, it accepts r only after r has arrived at both c_1 and c_2.

It is easily seen that the stored directions define a directed spanning tree T of the array, rooted at c_0. The reply signal r is initiated by the twigs of T, and passes through a node of T only when it has been received by all sons of that node. Thus the time required for r to reach c_0 is equal to the longest distance from c_0 to any twig of T, i.e., the longest distance from c_0 to any point of Σ, which is at most the diameter of Σ.//

To apply this result to proving the analog of Proposition 3.4.1, we change the reply signal from r to r' if it ever encounters a non-accepting cell. Thus if c_0 receives r, every cell of the simulation has accepted, and c_0 can accept; while if c_0 receives r', it can try again later (the simulation proceeds concurrently with the signal and reply propagation). If every cell in the simulation does eventually accept, c_0 will eventually receive r (since accepting states never change), and will accept*.

The analogs of Propositions 3.4.1-4 shows that the three definitions of acceptance are still equivalent even for connected arrays. We shall use the upper-left definition from now on, and shall always assume that c_0 is uniquely marked. The <u>language</u> $L(C)$ of arrays accepted by C, and the class $L_{(D)(B)C}$ of all languages accepted by (D)(B)CA's, are defined just as in the rectangular case (Section 5.3.1).

---

*The same argument is used to prove that a DBCA can verify its rectangularity: the reply r becomes r' if it encounters a concave corner; if c_0 receives r, there are no concave corners. Note that a DBCA could also verify rectangularity by simulating the proof of Proposition 4.2.1, but this would be very slow.

5.5 Speed comparisons

As we have just seen, CA's can simulate TA's in essentially real time, so that any language can be accepted by a CA at least as fast as it can be accepted by a TA (ignoring the acceptance propagation steps). On the other hand, some languages can be accepted by CA's much faster than by TA's. In this section we discuss comparative acceptance speeds for two-dimensional TA's and CA's.

Let $|\Sigma|$ denote the number of elements in the array Σ, and let $<\Sigma>$ denote the number of border points of Σ. If Σ is rectangular, say m by n, we have $|\Sigma| = mn$ and $<\Sigma> = 2(m+n-2)$ (provided $m, n \geq 2$); it follows that $|\Sigma| < <\Sigma>^2 < 16|\Sigma|$. In the nonrectangular case, let $R(\Sigma)$ be the framing rectangle of Σ (see the appendix to Section 5.3.1), and suppose that $R(\Sigma)$ is h by w. Then Σ has at least two border points (run ends) in each row of $R(\Sigma)$, and at least two in each column of Σ, so that $<\Sigma> \geq \max(2h,2w) \geq h+w$. On the other hand, $|\Sigma| \leq hw$, so that here again we have $|\Sigma| < <\Sigma>^2$. (In this case we cannot expect to show $<\Sigma>^2 < k|\Sigma|$ for any k, since $<\Sigma>$ might be as big as $|\Sigma|$.)

For any acceptor A (or C), suppose that for all Σ there exists a function f, taking natural numbers into natural numbers, such that if A accepts Σ at all, it does so in at most $f(<\Sigma>)$ time steps. We then say that A accepts in <u>perimeter time of order f</u>. For example, if f is linear, we say that A accepts in perimeter time; if f is quadratic, we say that A accepts in perimeter-squared time; and so on. Similarly, if there exists a function f such that A accepts any Σ, if at all, in at most $f(<\Sigma>)$ time steps, we say that A accepts in area time of order f. Note that by the remarks in the preceding paragraph, if A accepts in area time, it accepts in perimeter-squared time (and only if, in the rectangular case).

It is easily seen that, except in certain special cases, no TA can accept in faster than area time, and no CA can accept in faster than perimeter time. Indeed, if A accepts Σ in $t < |\Sigma|$ time steps, it cannot have seen all of Σ, and so accepts any array that matches Σ in at most t positions, and is otherwise arbitrary. Similarly, if C accepts Σ in $t < <\Sigma>/2$ time steps, the state of c_0 at step t cannot depend on the initial states of cells farther than t (in city block distance) away. For an m-by-n rectangular array, the farthest point from c_0 (the lower right hand corner) is at distance $m+n-2 = <\Sigma>/2$; hence the state of c_0 at step t cannot depend on this point. Thus C accepts any array that matches Σ out to city block distance t from c_0, and is otherwise arbitrary*.

*For arbitrary connected arrays, the farthest point may be much less than $<\Sigma>$ away from c_0, so that nontrivial recognition of certain classes of such arrays in time of order less than $<\Sigma>$ may be possible; but as the rectangular case shows, there do exist classes of arrays for which $O(<\Sigma>)$ is a lower bound. On the other hand, as we saw in Section 2.8, the intrinsic diameter of any Σ (= the greatest possible length of a shortest path in Σ between any two points of Σ) is at most half the total perimeter of Σ; thus if acceptance time is $O(<\Sigma>)$, the state of c_0 can depend on the state of any cell in Σ, however distant, so that there is no class of arrays that requires a lower bound higher than $O(<\Sigma>)$.

We may thus conclude that for "interesting" array languages L, the time required to accept any $\Sigma \epsilon L$ must be at least $|\Sigma|$ for TA's, and at least $<\Sigma>/2$ for CA's (or $O(<\Sigma>)$, in the non-rectangular case). Note that the "speed limit" for acceptance by two-dimensional CA's is much less than that for TA's (in fact, the latter is proportional to the square of the former), unlike the case in one dimension where both speed limits were the same (namely, $|\sigma|$).

To illustrate acceptance in minimal time by TA's and CA's, we may again consider (as in Section 3.4.3) the language $L_{\overline{x}}$ consisting of all rectangular arrays in which a particular vocabulary symbol x never appears. A DTBFSA can accept $L_{\overline{x}}$ in area time by systematically scanning Σ row by row; if it finds an x, it stops and enters a non-accepting state; if it completes the scan without finding an x, it accepts. A BCA that accepts $L_{\overline{x}}$ is defined as follows: The bottom cell of each column initiates a signal y that propagates upward through non-x's, but is stopped by x's. The rightmost cell of the top row initiates a signal z that propagates leftward through y's. If z reaches c_0 (at the left end of the top row), it accepts. Evidently this happens iff. no column of Σ contains an x. The time required for acceptance, on an m-by-n rectangular array, is just n-1(for the propagation of the y's up the columns) +(m-1) (for the propagation of the z's across the top row), which is exactly $<\Sigma>/2$. Thus $L_{\overline{x}}$ is an example where CA's accept much faster than TA's.

Acceptance by CA's in perimeter time is of particular interest, since this is (on the order of) the fastest time possible. In the following subsections we give a number of examples of perimeter-time acceptance algorithms for CA's. These algorithms illustrate the value of two-dimensional CA's as fast devices for array recognition.

It is straightforward to establish analogs of the upper bounds on acceptance time, discussed at the end of Section 3.4.3, for two-dimensional TBA's and BCA's; the details will not be given here.

5.5.1 Some efficient acceptance algorithms for rectangular arrays

In this section we describe several perimeter-time algorithms for acceptance of specific rectangular array languages by DBCA's. [Note that for an m-by-n rectangle, the perimeter 2(m+n-2) is just twice the city block diameter.] Some perimeter-time algorithms for connected array languages will be presented in the next section.

In Section 3.4.4 we gave a number of efficient algorithms for string language acceptance by one-dimensional DBCA's. These algorithms required times on the order of the string length. As an immediate application of these algorithms to rectangular arrays, consider the set of arrays each of whose rows belongs to a given string language, e.g., to the palindromes or well-formed parenthesis strings. We can accept such a set of arrays in O(perimeter) time as follows:

a) Each row simulates a one-dimensional DBCA, with distinguished cell at its left end, that accepts iff. the row belongs to the given string language. This requires O(array width = string length) time.

b) The left-hand column simulates a one-dimensional DBCA, with distinguished cell at the top, that accepts iff. the entire column consists of accepting states of the row DBCA's. [Once the left cell of a row DBCA accepts, that cell marks itself permanently, and the column DBCA accepts iff. the entire column is marked.] This can easily be done in O(array height) time, once the rows have all accepted. [The bottom left cell, when its row has accepted, initiates a signal that travels upward through acceptance-marked cells; if it reaches the top, the top cell accepts.]

Thus the total acceptance time is O(width)+O(height)=O(diameter). An exactly analogous scheme can be used to accept arrays whose columns belong to a given string language.

We next consider languages defined in terms of the number(s) of occurrences of particular symbols. For example, let us consider arrays of 0's and 1's, and consider the language consisting of all such arrays in which the number of 1's is odd (or even), or in which the number of 1's equals (or exceeds) the number of 0's. In Section 3.4.3 we saw that the analogous one-dimensional languages could be accepted by converting the number of 1's to a binary number, stored in the leftmost $\log_2$(string length) cells, since we can then check whether the least significant bit is 0 or 1, to determine the parity of the number of 1's. Similarly, if we simultaneously convert both the number of 1's and the number of 0's to binary numbers, both stored in the leftmost cells (without interfering with one another), we can do a bit-by-bit comparison of these numbers to determine which of the numbers is greater, or whether they are equal. The binary conversion and comparison processes both take only O(length) time.

In two dimensions, we can similarly convert the number(s) of 1's (and 0's) on each row to a binary number. [We assume here that the rows are at least as long as the columns; this assures that the final binary numbers for the entire array, which are O($\log_2$(area)) bits long, will fit on the rows, since row length $\geq$ (area)$^{1/2}$ > $\log_2$(area). If the rows are shorter than the columns (which can be verified in O(perimeter) time; see the beginning of Section 5.5.2), we use an analogous process with the roles of columns and rows interchanged.] We must now add up these numbers to obtain a total sum, say on the top row. To do this, let us first mark the odd-numbered and even-numbered rows distinctively (this takes O(height) time, using a signal that travels from bottom to top and alternates between two values; see the beginning of Section 5.5.2). We now simultaneously add each even-numbered row to the odd-numbered row above it; this takes O(log width) time (see the next paragraph for the details). When the additions are complete, we shift the sums upward until they are packed into the upper half of the rows; this takes (height/2) time. We now again simultaneoulsy add even-numbered rows to the odd-numbered rows above them, which takes O(log width) time. The entire process is repeated, until finally the sums corresponding to the top and bottom halves of the array, in the top two rows, are added to yield the final sum. The sums never get longer than log(area) $\leq$ log(width^2) = 2log(width), which is O(log width). The total shifting time is height/2 + height/4 + $\cdots$ = O(height). The number of times additions are performed is O(log height), and the additions take O(log width); thus the total addition time is O(log height x log width) $\leq$ O($\log^2$width) < O(width). The entire processing time thus consists of O(width) for the row additions, O(height) for the upward shifting of rows, and < O(width) for the additions of rows -- a total of O(height) + O(width) = O(perimeter) time steps.

The addition of pairs of rows proceeds straightforwardly. Each cell (say the kth) on an odd-numbered row contains a pair of bits (α_k, β_k); initially α_k is the kth bit of the sum on that row, and $\beta_k = 0$. The cell below it on the even-numbered row also contains a pair of bits (α'_k, β'_k) with $\beta'_k = 0$. At the first step, the cells on the even rows go into "blank" states. The new states of the cells on the odd rows are determined as follows:

Old α_k	α'_k	New α_k	New β_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

At subsequent steps, the new states of the odd-row cells are determined analogously:

Old α_k	Old β_{k-1}	New α_k	New β_k
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

To insure that the additions are all done simultaneously as soon as the rows are packed, we can assume that the lowest non-blank row is specially marked; this is trivial to do initially, since the bottom row has #s below it, and the marks can be inherited each time the lowest non-blank row is added to the row above it. That row can initiate a signal that propagates upward through non-blanks. When the signal reaches the top row, it knows that the packing is complete, and can initiate a synchronization process that causes all the pairs of rows to add at once. The signal propagation and synchronization steps, in preparation for the ith set of additions, take $O(\text{height}/2^{i-1})$ time, the same as the time for the preceding shifting and packing, so that our time order computation is not affected.

The new shifting and packing can begin as soon as the first step of the current addition is performed; but to insure that this addition is finished before the next one is initiated, we can proceed as follows: Assume that the cells containing bits of a sum are all marked, whether the bits are 1's or 0's; this can easily be done when the initial row sums are created, and can be preserved as part of the addition process. Concurrently with the addition, we can synchronize the marked cells on the lowest non-blank row; when the synchronization is done, we know that the addition is also (nearly) done, and can initiate the upward signal that will indicate completion of the packing. (A simpler method of counting 1's in a rectangular array is described in [14].)

As a generalization of counting 1's, we can count occurrences of any *given local pattern*, e.g., $\begin{matrix}1&1&1\\&1&\end{matrix}$. A uniquely defined cell (such as the upper left cell) of each such pattern first identifies itself; this can evidently be done by examining states of neighbors out to a bounded distance, and takes a bounded number of time steps. We then count the number of cells that have identified themselves in this way, and can then apply any desired tests or comparisons to these counts (e.g., parity, majority, etc.). We can also compute geometrical properties of the array that can be defined in terms of local patterns. For example, in an array of 0's and 1's, the set of 1's is a union of non-touching rectangles iff. it has no concave corners (compare Proposition 5.3.1), and this can be detected by marking concave corners and detecting the absence of a marked cell. Moreover, such a set of 1's is a single rectangle iff. it has exactly four convex corners, and this too can be easily determined by detecting and counting the convex corners. To tell whether the rectangles are (e.g.) squares, each northwest corner (= 1 with 1's as right and lower neighbors) can compare the numbers of upper and left-hand edge points of its rectangle (we mark these points and shift the marks leftward or upward (respectively) through 1's until they reach the corner; if they stop coming at the same time, that rectangle is a square). We can then mark the northwest corners of squares, and subsequently count them.

As another example, the genus of an array of 0's and 1's (essentially the number of connected components of 1's minus the number of components of 0's; see Section 2.7) is equal to V-E+F, where

V is the number of 1's

E is the number of $\begin{matrix}1\\1\end{matrix}$'s and 11's

F is the number of $\begin{matrix}11\\11\end{matrix}$'s

We can count these types of patterns (independently), and combine the counts (subtraction can be done analogously to addition) to compute the genus. We can thus, in particular, determine whether the array has some given genus, such as zero or one. All these processes take times proportional to the array perimeter or diameter. We can tell when the counting is complete by, e.g., sending a signal from the lower right corner; when it reaches the upper left corner, we know that diameter time (or some multiple of it, if we use a slow signal) has passed.

It is more complicated to count the connected components in an array of 0's and 1's. In the following paragraphs we describe a method, based on [15-16], of doing this in O(perimeter) time. The method makes use of a "shrinking" operation that reduces each component of 0's or 1's to a single point. We first describe this operation, and then show how it can be used to count components of 1's or 0's -- for example, to determine whether or not the 1's are connected.

Let Ψ denote the operation that

a) Changes a cell from 1 to 0 if its right and lower neighbors are both 0's

b) Changes a cell from 0 to 1 if its right, lower, and lower-right diagonal neighbots are all 1's

and leaves all other cells unchanged. In other words, Ψ deletes upper-left convex corners from the set S of 1's, and fills upper-left concave corners. If we are given the neighborhood

PA
BC

of the point P, then we change P from 1 to 0 iff. A=B=1, and change it from 0 to 1 iff. A=B=C=1. Equivalently, it is not hard to show that

$$\Psi(P)=1 \text{ iff. } P+A=2,\ P+B=2, \text{ or } A+B+C=3$$

Note that Ψ(P) depends only on two neighbors and one diagonal neighbor of P; it requires two time steps to compute. We could have defined Ψ using other pairs of neighbors (right and upper, left and upper, left and lower) rather than the right and lower neighbors; analogous results can be obtained for these alternative definitions.

We will next show that, in a sense to be defined more precisely below, applying Ψ to an array of 0's and 1's preserves the connectedness of both the 0's and the 1's. Our definition of Ψ assumes that we use 4-connectedness for the 1's and 8-connectedness for the 0's. If we want to use the opposite types of connectedness, we should define Ψ to change P from 1 to 0 iff. $A=B=C=0$, and to change P from 0 to 1 iff. $A=B=1$. Readily, this is equivalent to $\Psi(P)=1$ iff. $P+C=2$ or $P+A+B\geq 2$.

For brevity, denote $\Psi(P)$ by P'. Let S and S' be the sets of 1's before and after a given application of Ψ, and let $\overline{S}$ and $\overline{S}'$ similarly be the sets of 0's. For any set T, let T_1' and T_0' be the sets of points of S' and of $\overline{S}'$, respectively, that either lie in T, or have points of T as right and lower neighbors. We shall now prove that if U is a component of S, then U_1' is a component of S', and if V is a component of $\overline{S}$, then V_0' is a component of $\overline{S}'$. [We assume here that U and V consist of more than one point; if not, it is clear that these points disappear under Ψ, and that U_1' and V_0' are empty.]

Note first that if P' is in U_1' but not in U, we have $P'=1$ but $P=0$ (since P has points of U as right and lower neighbors, if it were 1 it too would be in U, since U is a component of 1's). Hence P must have a neighborhood $\begin{matrix}PA\\BC\end{matrix}$ in which $A=B=C=1$. Since $A=1$ and $C=1$, we must have $A'=1$, and similarly $B'=1$; thus P' has neighbors A' and B' that are both in U_1' and U. It follows that to prove the connectedness of U_1', we need only show that any two points that are in both U and U_1' have a path joining them in U_1'.

Let Q,R be two such points; let ρ be a path joining them in U (ρ exists since U is connected) that is as short as possible; and let X be any point on ρ such that $X'=0$. (If there is no such X, ρ is a path in U_1' and we are done.) Thus $X=1$ and $X'=0$, so that the neighborhood of X is of the form

$$\begin{array}{ccc} A & B & \\ C & X & 0 \\ & 0 & \end{array}$$

Since ρ is as short as possible, it must pass through both B and C; hence B,C are in U, so that $B=C=1$. It follows that $A'=B'=C'=1$, so that A',B',C' are all in U_1'. We can thus replace X by A on ρ, and by doing this for every such X, we obtain a path ρ' in U_1' from Q to R, as desired. Thus U_1' is connected.

We next show that U_1' is a component of S' -- or, equivalently, that any point of S' adjacent to U_1' must be in U_1'. Suppose first that the *given point* P is in both S and S', i.e., $P=P'=1$. If P is adjacent to U, it is in U, hence in U_1', and we are done. Otherwise, P must be adjacent to some $Q\in\overline{S}$ for which $Q'\in S'$ is in U_1'. The neighborhood of P must thus look like

$$\begin{array}{cc} P & R \\ Q & A \\ B & C \end{array} \quad \text{or} \quad \begin{array}{ccc} P & Q & A \\ R & B & C \end{array}$$

where A,B,C are in U (they are all in S since $Q=0$ but $Q'=1$; and A,B are in U since Q' is in U_1'). Since $Q=0$ but $P'=1$, we cannot have $R=0$; but if $R=1$, P is connected to U and so is in U. Similarly, if P is in $\overline{S}$ and S', i.e., $P=0$ but $P'=1$, then P has a neighborhood of the form

$$\begin{array}{ccc} & X & Y \\ Z & P & A \\ W & B & C \end{array}$$

with A,B,C in S. If A or B is in U_1', then it is also in U, so that P is in U_1'. Otherwise, X or Z must be in U_1', and if $P \not\in U_1'$ this means that W or Y must be in U; but then A,B,C are in U, so that P is in U_1', contradiction. Thus U_1' is a component.

Note finally that if U has more than one point, U_1' cannot be empty; for example, if P is the leftmost of the uppermost points of U, then $P'=1$ (since either its right or lower neighbor is in S), so that P is in U_1'. [By the same argument, if V is a component of $\overline{S}$ and has more than one point, V_0' cannot be empty.]

We now turn to the proof that if V is a component of $\overline{S}$, than V_0' is a component of $\overline{S}'$. Let ρ be a path in V that is as short as possible, and let $\hat{\rho}$ be a run of points of ρ that are not in $\overline{S}'$, but such that the points of ρ immediately preceding and following $\hat{\rho}$ are in $\overline{S}'$. Then the neighborhood of $\hat{\rho}$ must be of the form (e.g.)

$$\begin{array}{ccccc}
 & & A & B & C \\
 & & D & \hat{\rho} & s \\
E & F & \hat{\rho} & s & s \\
G & \hat{\rho} & s & s & \\
H & s & s & &
\end{array}$$

where the s's are in S. Thus D and F are in $\overline{S}'$, so that if the point of ρ preceding $\hat{\rho}$ is E or G, and the point following $\hat{\rho}$ is A or B, we can replace $\hat{\rho}$ by the subpath F,D, which lies in $\overline{S}'$. On the other hand, if the preceding point is H, we have $G' \in \overline{S}'$, and if the following point is C, we have $B' \in \overline{S}'$, so that in these cases $\hat{\rho}$ can be replaced by G,F,D, by F,D,B, or by

G,F,D,B, all in $\bar{S}'$. It follows that any (shortest) path in V can be converted into a path in V_0'. Thus any two points in V_0' are joined by a path in V_0', so that V_0' is connected.

To show that V_0' is a component, we prove that any point of $\bar{S}'$ adjacent to V_0' must be in it. Let the given point P be in S; its neighborhood must then be

$$\begin{array}{ccc} & X & \\ Y & P & A \\ & B & \end{array}$$

where A,B are in $\bar{S}$. If A or B is in V_0', they are both in V, so that P is in V_0'. Otherwise, X or Y must be in V_0', say X, so that $X'=0$; but since $P=1$, this means that $X=0$, so that X is in V, and this implies A,B in V, proving P V_0'. On the other hand, let P be in $\bar{S}$; then if it is adjacent to V, it is in V, hence in V_0', and we are done. Suppose it is adjacent to a point Q of V_0' that was not in V, so that its neighborhood looks like

$$\begin{array}{cc} P & \\ Q & A \\ B & \end{array} \qquad \text{or} \qquad \begin{array}{ccc} P & Q & A \\ & B & \end{array}$$

where A and B are in V; then P is also in V, hence in V_0'.

Finally, we show that if U is adjacent to V, then U_1' is adjacent to V_0'. We recall (Section 2.6) that U adjacent to V implies that U surrounds V or vice versa; suppose the former. Let P be the leftmost of the uppermost points of V, so that P's upper neighbor Q is in U. Then readily $P' \in S'$ and $Q' \in \bar{S}'$, so that $P' \in U_1'$ and $Q' \in V_0'$.

For any component U of S, let x_u and y_u be the coordinates of the leftmost row and uppermost column that contain points of U. Let P be any point of U at maximum city block distance from (x_u,y_u). Thus the right and lower neighbors of P cannot be in U, hence are in $\overline{S}$, so that P'=0. Thus applying Ψ decreases the maximum city block distance of U from (x_u,y_u).

No point of U_1' can lie to the left of x_u or above y_u, since a point to the left of x_u cannot be in u or have its lower neighbor in U, and a point above y_u cannot be in U or have its right neighbor in U. On the other hand, any leftmost uppermost or uppermost leftmost point of U must be in U_1' (as long as U has more than one point), since such a point must have either its right or lower neighbor in U. Thus x_u and y_u are still the leftmost and uppermost coordinates for U_1', as they were for U.

The remarks in the last two paragraphs imply that when Ψ is repeatedly applied, the leftmost and uppermost coordinates (x_u,y_u) of any U remain unchanged, but the city block distance of U from (x_u,y_u) keeps decreasing. This means that U must eventually shrink to a single point, which must evidently be the point (x_u,y_u), and this point then vanishes when Ψ is applied again. The number of steps required for this to happen is at most (twice*) the original maximum city block distance of U from (x_u,y_u). Analogous remarks apply to any component V of $\overline{S}$. [The only exception is the background component, which is not surrounded by any component of U.] Note that the time

*Since Ψ takes two time steps to apply; we ignore this factor of 2 below.

required for a component C to shrink to a point is at most the diameter of C's framing rectangle, which is less than the diameter of Σ.

Using Ψ, we can design a DBCA that counts connected components of 1's or 0's. We apply Ψ repeatedly; this shrinks every connected component, except for the background component of 0's, to a single point (which then vanishes), in time less than the diameter of Σ. When single-point components are created (or initially present), we can easily detect them, since they are 1's all four of whose neighbors are 0's, or 0's all eight of whose neighbors are 1's. Thus at the same time that Ψ annihilates them, we can replace them by special marks, say I and O, which can occupy a cell *without interfering* with the processing of the 1's and 0's by Ψ. We can now count these special marks, as described earlier in this section: add them up in each row, then add the row sums. [We know that O(diameter) time steps suffice for the I's and O's to have all been created, and to have been shifted to the left ends of their rows and counted there; thus after O(diameter) time steps we can safely initiate the process of adding the row sums.] This gives us counts of the numbers of components of 1's and 0's in the original Σ (except for the background component of 0's). Thus we can accept Σ iff., e.g., these numbers have given values or given parities, e.g., if they 1's are connected, or if there are no holes in the 1's (= non-background components of 0's). [Acceptance based on comparison of these numbers, e.g., more components than holes, can be done without using Ψ, since the difference between the number of components of 1's and the number of non-background components of 0's is just the genus, which can be computed more simply.]

We can also use Ψ to define DBCA acceptors for various other languages. For example, to accept iff. the 1's consist of a single closed curve, we can first check the fact that every 1 has exactly two 1's as neighbors, which implies that the 1's are a union of non-touching simple closed curves (Section 2.3); and we can then check that the 1's are connected (or equivalently, that they have exactly one hole).

5.5.2 Some efficient acceptance algorithms for connected arrays

In this section we present some perimeter-time algorithms for DBCA acceptance of specific connected array languages. In particular, we may consider languages defined by the geometrical properties of the array Σ (of non-#s) itself. As one example, in Section 5.3 we showed how a DBCA can accept Σ iff. it is rectangular. It is also easy to define DBCA acceptance algorithms for rectangles having particular properties, e.g., side lengths odd or even, height equal to (or greater than) width, etc. For example, to detect evenness of rectangle width, the upper right corner sends a signal to the upper left corner (c_0), and the signal alternates between two states, say α and β (ie., if a cell's right neighbor is α, the cell becomes β, and vice versa), where the initial signal is α; if c_0 receives β, it accepts. To detect the fact that height is greater than width, the lower right corner sends a signal to c_0 that travels diagonally, i.e., alternately upward and leftward (if a cell's lower neighbor is α, the cell becomes β; if a cell's right-hand neighbor is β, the cell becomes α). If an α has a # on its left, it becomes a γ; this γ moves upward one step and becomes δ; δ moves upward until it reaches c_0, which then accepts. If γ reaches c_0, the height and width are equal.

To determine whether or not Σ is simply connected (i.e., has no holes), we can proceed as follows [4]: In the proof of Theorem 5.4. we showed that the distinguished cell

c_0 of Σ can identify itself in time on the order of the outer perimeter of Σ. (The processes on inner borders may still be going on, but they will not lead to anything, and can safely be ignored.) The outer border of Σ then marks itself by simulating a border-following process starting from c_0; this again takes outer perimter time. [A cell c may be on both the outer border and an inner border, but this can be detected by checking whether the BF simulation has made use of all the # neighbors of c; if not, the mark is modified to indicate that c is on more than one border.] After the marking is completed, each outer border cell c_1 sends out a signal that moves rightward along its row until it reaches another border cell c_2 and hits a #; this must happen in a number of time steps at most equal to the width of Σ, which is evidently less than half the outer perimeter. If c_2 is marked (i.e., is on the outer border), the signal is erased. If c_2 is not marked, or has a modified mark, the signal generates a reply that moves back along the row until it reaches c_1. [If Σ has a hole, this must happen. Indeed, let P be a hole point that is as far to the left as possible; thus P's left neighbor Q is a hole border point. If we move leftward from Q through non-#s until we hit a #, the last non-# must be an outer border point, since there are no hole points to the left of P. Thus if Σ has a hole, some outer border point must have the property that when we move rightward from it through non-#s until we hit a #, the last non-# must be a hole border point.] When this happens (or if c_1 had a modified mark to begin with), c_1 generates a signal that travels around the outer border to c_0. If c_0 receives such a signal, it knows that Σ is not simply connected. If it fails to receive such a signal within (say) twice outer perimeter time after the marking of the outer border was completed (it can determine this time by sending a half-speed signal around the outer border), it knows that Σ is simply connected, and can accept. The entire process takes O(outer perimter) time.

To determine whether Σ is a simple arc or simple closed curve, c_0 identifies itself and marks the outer border. When this is finished, each outer border cell verifies that it has no unmarked neighbors, and at most two marked neighbors. If a cell fails to verify this, it sends a signal to c_0, which then knows that Σ is not an arc or curve. If this signal does not arrive within outer perimter time, c_0 knows that Σ is an arc or curve. If two border cells have only one neighbor, Σ is an arc; if all of them have two neighbors, Σ is a curve.

As we saw in the proof of Theorem 5.4.2, the distinguished cell c_0 of a DBCA can send out a signal and receive a reply from every cell in O(diameter) time, hence in O(perimeter) time. This reply can inform c_0 about the presence or absence of specific symbols, or local patterns, in Σ. We can also determine (e.g.) the parity of the number of these symbols; in fact, each cell can receive parity information from its children on the spanning tree and compute its own parity (i.e., the parity corresponding to the set of cells below it in the tree) for transmittal to its own parent.

Counting specified symbols (or local patterns) is more difficult. For a non-BCA there would be no problem; we could simply construct a containing rectangle and use the methods of Section 5.5.1 to count the various types of non-#s in that rectangle. For a BCA, we can use a different algorithm that once again involves constructing a spanning tree. In such a tree, we know that each node has degree (number of non-# sons) ≤ 3, since a node has only four neighbors, and one of them must be its father node. [The root node of the tree has no father, but since it is the upper left corner of Σ, it has at most two non-# sons.] In the following paragraphs we describe an algorithm in which each node of the tree successively outputs (to its father node) the base-4 digits of the number of nodes in its subtree, least significant digit first. The root node, which has no father, can shift these digits down the tree so that they are stored along a longest path*. [Let h, the tree height, be the length of such a longest path; then readily the total number of nodes in the tree is at most $(1+3+3^2+\cdots+3^h) < 3^{h+1}$, so that there is room to store this number along the path.]

---

*This shifting is understood to be independent of, and not to interfere with, the upward shifting of the digits to be described below.

The basic idea of the algorithm is as follows: We assume that the tree has been constructed, so that each cell knows which of its neighbors are its father and its sons. The states of a cell are triples of the form (α,β,γ), where

$\alpha = 0, 1, 2$ or 3 represents a sum digit (to base 4)

$\beta = 0, 1, 2$ or 3 represents a carry digit. [Since a cell has at most three non-# sons, each of which is sending it a base-4 digit (≤ 3), and since its own digit (the carry digit from the previous addition) is also ≤ 3, it can add these digits and obtain a sum $\leq 4 \cdot 3 = 12 = 3 \cdot 4^1 + 4^0$; this sum can thus be represented using a pair of base-4 digits, $\beta \cdot 4^1 + \alpha \cdot 4^0$.]

γ = indicates the type of tree node and the action to be taken with respect to the addition process; the significance of γ will become clear in the course of describing the algorithm.

At the first step, each cell changes to state $(1,0,t)$ if it is a twig node, and $(1,0,\bar{t})$ if it is not. At each subsequent step,

1) A cell in state $(1,0,\bar{t})$ remains unchanged unless all its sons are in states with third terms t or v; it then changes to $(1,0,u)$. [Informally, the u indicates that the cell is ready to add its sons' inputs; the v indicates a cell that has just added its sons' inputs.]

2) A cell in state (s,0,u) changes to state (q,p,v), where 4p+q = s+ (the sum of the digits shifted up from the cell's sons). [It can be shown, by induction on tree height, that whenever a cell c is in a state with third term u, its sons are all in states with third terms t or v, so that they all have sum digits that can be shifted up to c for addition. It can also be shown that under these circumstances the second (carry digit) term of c's state must be zero.]

3) A twig node cell (in state (1,0,t)) shifts its sum digit (1) up whenever its father has third term u. [It can be shown that this happens only once.]

4) A cell in state (s,c,v) remains unchanged unless its father is in a state with third term u (or unless it is the root node); it then shifts s up to its father (or down along the storage path, if it is the root node), and changes to state (c,0,u).

Thus u serves as a signal to a cell's sons that they can shift their sum digits up for addition (2-4). This insures that all sons shift up at the same time, even though the information may have arrived at the sons at different times (because their subtrees have different heights). Note that the cell cannot be in a u state until all its sons are ready (1). The v symbol serves as a signal to a cell's father that the cell is ready to shift its sum digit up; at the same time, it serves as a signal to the cell's sons that the cell is not ready for them to shift their digits up to it. When a cell shifts a digit up to its father, its own state changes from v to u (4), indicating to its sons that it is ready to accept their digits again. Thus as long as digits remain to be shifted up to a cell, its state alternates between u and v.

It is not hard to prove* that a cell at height $h \geq 1$ in the tree remains in state (1,0,t) until step h+1, when it changes to state (1,0,u). At the end of that step, all the cell's sons' states have third terms t or v. [In general, whenever a cell's third term is u, its sons' third terms are all t or v.] Thus the cell now adds its sons' outputs; by induction hypothesis, these are the least significant digits (to base 4) of the numbers of nodes in the sons' subtrees, so that the cell's sum digit is now the least significant digit of the sum of these sums, i.e., of the number of nodes in its own subtree. When the cell's father is ready, the cell outputs this digit, changes its carry digit to a sum digit, and becomes ready to accept further inputs from the sons. These can immediately output to it the next least significant digits of their subtrees' numbers of nodes. The cell adds these to its own sum digit, which yields the next least significant digit of its own subtree's number. This process continues until the

*For the details see A. Wu, Cellular Graph Automata, 2, University of Maryland Computer Science Center Technical Report 621, December 1977, Section 1.3.3.

sons have no further information to transmit. It can be shown that this takes 3h+3 time steps: h+1 until the additions start, and two steps per digit (alternating between u and v states) until they are finished. Thus the total time required for the sum to be computed for the entire tree is about three times the distance from the root cell to the (twig) cell farthest from it, which is on the order of the diameter of Σ. It is easy for the root cell to know when this process has been completed (e.g., use a signal that travels to a farthest cell at half speed and then back to the root at full speed; when the return signal arrives, the elapsed time is essentially 3h.

It is trivial to modify this algorithm to count the number of cells having a particular state, rather than all the cells; or the number of occurrences of a particular local pattern (a uniquely defined cell in each such occurrence first goes into a special state, and these states are then counted).

References

1. J. E. Hopcroft and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, MA, 1969.

2. F. R. Moore and G. G. Langdon, A generalized firing squad problem, Info. Control 12, 212-220

3. A. R. Smith III, Cellular automata and formal languages, Proc. 11th SWAT, 1970, 216-224.

4. A. R. Smith III, Two-dimensional formal languages and pattern recognition by cellular automata, Proc. 12th SWAT, 1971, 144-152.

5. M. Blum and C. Hewitt, Automata on a 2-dimensional tape, Proc. 8th SWAT, 1967, 155-160.

6. D. L. Milgram, A region crossing problem for array-bounded automata, Info. Control 31, 1976, 147-152.

7. D. L. Milgram, Web automata, Info. Control 29, 1975, 162-184.

8. J. Mylopoulos, On the recognition of topological invariants by 4-way finite automata, Computer Graphics Image Processing 1, 1972, 308-316.

9. D. L. Milgram and A. Rosenfeld, Array automata and array grammars, IFIP Congress 71, North-Holland, Amsterdam, 1972, 69-74.

10. A. Rosenfeld, Some notes on finite-state picture languages, Info. Control 31, 1976, 177-184.

11. A. R. Smith III, Cellular automata complexity trade-offs, Info. Control 18, 1971, 466-482.

12. H. B. Nguyen and V. C. Hamacher, Pattern synchronization in two-dimensional cellular spaces, Info. Control 26, 1974, 12-23.

13. I. Shinahr, Two- and three-dimensional firing squad synchronization problems, Info. Control 24, 1974, 163-180.

14. S. R. Kosaraju, On some open problems in the theory of cellular automata, IEEE Trans. Computers 23, 1974, 561-565.

15. W. T. Beyer, Recognition of topological invariants by iterative arrays, MAC TR-66, MIT, Cambridge, MA, 1969.

16. S. Levialdi, On shrinking binary picture patterns, Comm. ACM 15, 1972, 7-10.